Network Manager IP Edition
4.2

Reference

2021-4213-01

Note

Before using this information and the product it supports, read the information in "Notices" on page 995.

This edition applies to version 4.2 of IBM Tivoli Network Manager IP Edition (product number 5724-S45) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this publication

The *IBM Tivoli Network Manager Reference* contains reference information including the system languages, databases, and Perl API used by Network Manager. This publication is for advanced users who need to customize the operation of Network Manager.

Publications

This section lists publications in the Network Manager library and related documents. The section also describes how to access IBM publications online and how to order publications.

Your Network Manager library

The following documents are available in the Network Manager library:

- The *IBM Tivoli Network Manager IP Edition Release Notes* give important and late-breaking information about Network Manager. This publication is for deployers and administrators, and should be read first.
- The *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*describes how to install Network Manager. It also describes necessary and optional post-installation configuration tasks. This publication is for administrators who need to install and set up Network Manager.
- The *IBM Tivoli Network Manager IP Edition Administration Guide* describes administration tasks such as how to start and stop the product, discover the network, poll the network, manage events, administer processes, and query databases. This publication is for administrators who are responsible for the maintenance and availability of Network Manager.
- The *IBM Tivoli Network Manager Reference* contains reference information including the system languages, databases, and Perl API used by Network Manager. This publication is for advanced users who need to customize the operation of Network Manager.

Prerequisite publications

To use the information in this publication effectively, you must have some prerequisite knowledge, which you can obtain from the following publications:

- *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*

  Includes installation and upgrade procedures and describes how to configure security and component communications. The publication also includes examples of Tivoli Netcool/OMNIbus architectures and describes how to implement them.
- *IBM Tivoli Netcool/OMNIbus User's Guide*

  Provides an overview of the desktop tools and describes the operator tasks related to event management using these tools.
- *IBM Tivoli Netcool/OMNIbus Administration Guide*

  Describes how to perform administrative tasks using the Tivoli Netcool/OMNIbus Administrator GUI, command-line tools, and process control. The publication also contains descriptions and examples of ObjectServer SQL syntax and automations.
- *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*

  Contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands.
- *IBM Tivoli Netcool/OMNIbus Web GUI Administration and User's Guide*

  Describes how to perform administrative and event visualization tasks using the Tivoli Netcool/OMNIbus Web GUI.

Accessing terminology online

The IBM Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:

http://www.ibm.com/software/globalization/terminology

Accessing publications online

IBM posts publications for this and all other products, as they become available and whenever they are updated, to the IBM Knowledge Center Web site at:

http://www.ibm.com/support/knowledgecenter/

Network Manager documentation is located under the **Cloud & Smarter Infrastructure** node on that Web site.

Note: If you print PDF documents on other than letter-sized paper, set the option in the **File** > **Print** window that allows your PDF reading application to print letter-sized pages on your local paper.

Ordering publications

You can order many IBM publications online at the following Web site:

http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, contact your software account representative to order IBM publications. To locate the telephone number of your local representative, perform the following steps:

1. Go to the following Web site:

   http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss
2. Select your country from the list and click **Go**. The **Welcome to the IBM Publications Center** page is displayed for your country.
3. On the left side of the page, click **About this site** to see an information page that includes the telephone number of your local representative.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully.

Accessibility features

Network Manager includes the following major accessibility features:

- Operations that use a screen reader.

Network Manager uses IBM Installation Manager to install the product. You can read about the accessibility features for IBM Installation Manager at https://www.ibm.com/support/knowledgecenter/SSDV2W/im_family_welcome.html.

Network Manager uses the latest W3C Standard, http://www.w3.org/TR/wai-aria/, to ensure compliance to http://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards/section-508-standards), and http://www.w3.org/TR/WCAG20/. To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Network Manager online product documentation in IBM Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at https://www.ibm.com/support/knowledgecenter/v1/content/about/releasenotes.html#accessibility.

Keyboard navigation

This product uses standard navigation keys.

Interface information

Network Manager provides the following features suitable for low vision users:

- All non-text content used in the GUI has associated alternative text.
- Low-vision users can adjust the system display settings, including high contrast mode, and can control the font sizes using the browser settings.
- Color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.

Network Manager provides the following features suitable for photosensitive epileptic users:

- The Network Manager user interfaces do not have content that flashes more than two times in any one second period.

The Network Manager web user interface includes WAI-ARIA navigational landmarks that you can use to quickly navigate to functional areas in the application.

Extra steps to configure Internet Explorer for accessibility

If you are using Internet Explorer as your web browser, you might need to perform extra configuration steps to enable accessibility features.

To enable high contrast mode, complete the following steps:

1. Click **Tools** > **Internet Options** > **Accessibility**.
2. Select all the check boxes in the Formatting section.

If clicking **View** > **Text Size** > **Largest** does not increase the font size, click **Ctrl +** and **Ctrl -**.

Related accessibility information

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service
800-IBM-3383 (800-426-3383)
(within North America)

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see https://www.ibm.com/able.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education Web site:

https://www.ibm.com/training/search?query=tivoli

Support and community information

Use IBM Support, Service Management Connect, and Tivoli user groups to connect with IBM and get the help and information you need.

IBM Support

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

Online
Go to the IBM Software Support site at https://www.ibm.com/support/home/ and follow the instructions.

IBM Support Assistant
The IBM Support Assistant (ISA) is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. The ISA provides quick access to support-related information and serviceability tools for problem determination. To install the ISA software, go to https://www.ibm.com/support/knowledgecenter/SSLLVC/welcome/isa_welcome.html

Part 1. Languages

Network Manager uses different languages, such as the Object Query Language, stitcher language, and the syntax of the AOC files.

Chapter 1. Object Query Language

Object Query Language (OQL) is a version of the Structured Query Language (SQL) that has been designed for use in Network Manager. The components create and interact with their databases using OQL.

Use OQL to create new databases or insert data into existing databases (to configure the operation of Network Manager components) by amending the component schema files. You can also issue OQL statements using the OQL Service Provider, for example, to create or modify databases, insert data into databases and retrieve data.

For more information about the OQL schema used by Network Manager, see the *IBM Tivoli Network Manager Reference*.

The OQL Service Provider is described in the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Conventions

The following conventions have been used to explain the OQL syntax:

- OQL keywords and their required punctuation are shown in **bold** in examples.
- Parameters are shown in italics.
- Optional parameters are shown enclosed by square brackets [].
- OQL commands are terminated with a semicolon.
- An ellipsis (...) following a list of parameters indicates that you can continue the list if necessary.
- OQL Service Provider command lines are shown with the example command line prompt | `phoenix:1.>` followed by relevant sample output.
- The system name within the prompt appears as `phoenix` within this manual. This is replaced by an appropriate value for your system, such as the host name.

Sample databases

The following tables contain data in the staff databases.

Table 1. Data in the staff.managers database table

EmployeeID	Name	Department	Gender	Age
1	Matt	Development	M	28
2	Irene	Customer Services	F	52
3	Ernie	Sales	M	23
4	Paul	Marketing	M	26
5	Jim	Support	M	27

Table 2. Data in the staff.employees database table				
EmployeeID	**Name**	**Skills**	**Gender**	**Age**
6	Paul	HTML, C++, Java™	M	26
7	Carl	Perl	M	32
8	Rob	UNIX, Perl	M	23
9	Sarah	Java, C++	F	24
10	Lisa	UNIX, HTML	F	25

Table 3. Data in the staff.contractors database table				
EmployeeID	**Name**	**Gender**	**Age**	**ExtraInfo**
11	James	M	22	ContractLength = 6 months; Department = Marketing;
12	Karen	F	25	ContractLength = 5 months; Department = Sales;
13	Jane	F	26	ContractLength = 7 months; Department = Development;
14	Richard	M	23	ContractLength = 1 month; Department = Operations;
15	Glenn	M	28	ContractLength = 2 months; Department = Operations;

Features of OQL

The following topics describe the features of Object Query Language (OQL).

Related reference

Stitcher language comments

Comments are introduced by - - or //. If a comment requires a carriage return, the characters on the next line must also be commented out.

General rules of OQL

OQL has rules that must be applied to all statements.

The following rules apply to OQL statements:

- All complete statements must be terminated by a semi-colon.
- A list of entries in OQL is usually separated by commas but *not* terminated by a comma.
- Strings of text are enclosed by matching quotation marks. The rules for quotation mark usage are described below.

Quotes in OQL

In OQL the TEXT datatype must be enclosed by *matching* quotation marks (either single or double quotes).

The following example is a standard OQL insert into the CTRL services.inTray table, showing values of the TEXT data type enclosed in double quotes, an integer value that is not enclosed in any quotation marks, and a LIST OF TYPE TEXT that is enclosed in square brackets. The comma separated data within the list is enclosed in double quotes.

```
insert into services.inTray
(
        serviceName, binaryName, servicePath, domainName, argList, retryCount
)
values
(
        "ncp_disco", "ncp_disco", "/opt/netcool/precision/Solaris2/bin", "MYDOMAIN",
        [ "-domain" , "NCOMS" , "-latency" , "60000" ], 5
);
```

Related reference

Stitcher language building blocks
To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

OQL punctuation

OQL has a set of punctuation marks that you use to structure OQL statements.

The following table describes the punctuation used in the OQL syntax.

Table 4. Punctuation used in the OQL syntax

Symbol	Meaning
-	Subtraction, negative.
+	Addition, positive.
*	Multiplication or a wild card that matches any characters within its expression.
(	Encloses items in a list, for example, preceding the **insert into** statement.
)	Encloses items in a list, for example, following the **insert into** statement.
,	Separates entries in a list.
[ ]	Encloses items of the list datatype.
{ }	Encloses items of the object datatype.
.	Separates database, table and column names.
;	Terminates OQL statements.
~	Matches regular expressions.
->	Retrieves a sub value of an object.

Logical operators of OQL

These logical operators are used for OQL: AND and OR.

The following table describes the logical operators to use in OQL.

Table 5. Logical operators of OQL	
Keyword	**Brief description**
AND	Combines search conditions, all of which must be true for the condition to be passed.
OR	Combines search conditions, one of which must be true for the condition to be passed.

Precedence and association of operators

The rules for precedence and association of operators determine the grouping of operators with operands, and indicate the order in which the operators in an expression are executed.

For complex expressions, use parentheses to avoid ambiguity.

The following table describes the operators.

Table 6. OQL operators in order of decreasing precedence			
Operator	**Description**	**Associativity**	**Precedence**
-	Negative sign	Non-associative	1 (highest)
*	Multiplication	Left	2
/	Division	Left	2
OR	Logical OR	Left	3
AND	Logical AND	Left	4
NOT	Logical NOT	Left	5
=	Equal to	Left	6
<>	Not equal to	Left	6
<	Less than	Left	6
>	Greater than	Left	6
<=	Less than or equal to	Left	6
>=	Greater than or equal to	Left	6
+	Addition	Left	7
-	Subtraction	Left	7 (lowest)

Use of regular expressions

You can use regular expressions in OQL and in stitcher language code. Regular expressions are particularly useful for defining filters.

Regular expressions contain a series of characters that define a pattern of text to be matched—to make a filter more specialized, or general. For example, the regular expression `^AL[.]*` searches for all items beginning with AL. The filter condition `EntityName Like ^..N[.]*` filters for all devices which have an N as the third letter of their name, and `EntityName Like [.]*G` filters for all devices whose name ends with the letter G. The following table describes the most common characters used in regular expressions.

Table 7. Regular expression characters

Character	Description	Example
\	The backslash (or escape character) quotes the character after it, both special and ordinary.	Use the backslash to specify a `.` (normally a special character) in a file name, for example. To select all `.sys` files you would state, `^*\.sys$`, where the backslash specifies that the dot following it is actually a real dot, not just a character representing any single character.
.	The dot represents any single character.	A dot can be anything. If you want to select five letter device names that begin with T and end with R, you would state, `^T...R $`, where the three dots in the middle mean that the three middle letters of the word can be any letter.
*	Like the dot, an asterisk can represent any character. However, whereas the dot can only represent a single character, the asterisk represents anywhere from zero to an infinite amount of characters.	`*.*`, returns strings beginning with any combination and any amount of characters (the first asterisk), and can end with any combination and any amount of characters (the last asterisk). This selects every single string available.
$	The dollar sign at the end of a regular expression signifies the end of a line, and, therefore, any character immediately before it must be located at the end of the string. Anywhere else in a regular expression, it matches itself.	`[.]*G$` selects every string which ends in G, regardless of the number of characters or types of characters in the string.
^	A hat (circumflex) at the beginning of a regular expression means that it is the beginning of a line, and any characters immediately following it must be located at the very beginning of the string. Anywhere else in a regular expression, it matches itself.	`^AL[.]*` returns strings beginning with AL. `^..N[.]*` returns strings beginning (^) with any two characters and the third character is an N.
[set]	A set of characters in square parentheses matches any single character from a set.	`^[abc].[def]$` selects all three character strings that begin with either a or b or c and end in either d or e or f.

Key differences between OQL and SQL

OQL differs from SQL in several ways.

- OQL supports object referencing within tables. Objects can be nested within objects.
- Not all SQL keywords are supported within OQL. Keywords that are not relevant to Network Manager have been removed from the syntax.
- OQL can perform mathematical computations within OQL statements.

Database and table creation

You can create databases and tables with the create command.

You can issue the **create** command to create a database.

```
create database database_name ;
```

You can issue the create command to create a table. When you create a table, you must define all the columns of the table as well as a datatype, such as text or integer, and if applicable, the column constraints and any default values.

```
create table database_name.table_name
(
        column_name [ constraints ] [ default default ] ,
        [ column_name [ constraints ] [ default default ] , ]
        [ additional_columns ]
        [ unique ( column_name ) , ]
        [ counter ( column_name ) , ]
        [ timestamp ( column_name ) ]
);
```

The entries within the brackets are separated by commas, although the last entry is not terminated by a comma.

The following topics describe how to use the **create** command.

Datatypes

All columns must have an associated datatype that indicates the acceptable input for that column.

The following table describes the datatypes that are used in OQL:

Table 8. Datatypes of OQL

Datatype	Description
TEXT	Holds plain text.
INT	Holds integer values.
UINT	Holds a 32-bit unsigned integer value.
FLOAT	Holds decimal values.
LONG64	Holds a 64-bit numerical value.
ULONG64	Holds a 64-bit unsigned numerical value.
DATA	Holds opaque data, typically of the binary format.
LIST TYPE *datatype*	Holds a list of particular datatypes. The list is enclosed in square brackets, `[]`.
OBJECT TYPE *datatype*	Holds objects of particular datatypes. The object is enclosed in curly braces, `{}`. Objects hold a list of varbinds (name/value pairs).
TIME	Holds information pertaining to time.

Objects and varbinds

An object is a comma-separated list of varbinds enclosed by curly braces, whereas a varbind is a name/value pair, for example, `ifIndex=20` or `ifDescr="utp10/100"`.

You can perform operations on one of the varbinds in an object using the -> symbol.

Related reference

Example selection based on part of an object
Use this example to orient yourself when retrieving records from tables based on criteria that apply to parts of a table row.

Example of updating an object
Use this example to learn how to use the **update** keyword in an object.

External datatypes

It is possible to define additional datatypes that map integers to the possible values of a column; these datatypes are called *external datatypes*, because they are defined externally to the present schema.

Syntax

In order to use a datatype that has been defined in a previously loaded schema, the current schema must contain a statement of the following structure.

```
data type datatype is external datatype ;
```

Examples

The following examples show the datatype declaration.

```
data type boolean is external boolean
data type entityTypes is external entityTypes
```

After you have made these declarations you can use the datatypes `boolean` and `entityTypes` in the current schema, provided that they have also been defined in a previously loaded schema.

Column constraints

Column constraints are restrictions on the data that can be inserted into a given column.

The following table describes the column constraints. Any of these constraints can be applied to any column.

Table 9. Column constraints

Constraint	Description
NOT NULL	Indicates that the column must be assigned a value. NULL is not the same as zero or white space.

Table 9. Column constraints (continued)	
Constraint	**Description**
PRIMARY KEY	Denotes that the column is a primary key for the table. The primary keys can be used to join related tables in a multiple table query. The combination of data in the PRIMARY KEY columns of a given table must be unique, although the data in each individual column does not necessarily have to be unique. For example, the primary key might comprise the multiple fields of forename, surname, and age. The primary key is internally indexed and so provides faster query times, but slower insert and delete times. Unique and indexed fields are also internally indexed. List and object fields cannot be indexed, thus they cannot be set as primary keys, unique fields, nor as part of an index definition.

If multiple constraints are specified for a single column, NOT NULL must be specified before PRIMARY KEY.

Default values

A default value can be applied to a column when it is created with the default keyword. If an explicit value is not provided for that column when data is inserted, the specified default value is used.

Unique

The optional unique column constraint ensures that all entries in the specified column are unique.

The unique column is internally indexed and so provides faster query times, but slower insert and delete times.

List and object fields cannot be indexed, thus they cannot be set as unique fields, nor as part of an index definition.

Counter

The optional counter column constraint delegates the responsibility of counting duplicate records to the specified column.

If you attempt to insert a duplicate record into the table, the insertion of the duplicate entry is suppressed and the value of the specified column is incremented in the record that already exists.

Timestamp

The optional timestamp column constraint stores the date and time information for the creation of the record in the specified column of the database table.

The timestamp properties are:

- Format: YYYY-MM-DD HH:MM:SS.nnnnn....
- Range: 0001-01-01 00:00:00.....to 9999-12-31 23:59:59.999999.....

Examples of database and table creation

The following examples use the create keyword to create the sample `staff` database and define the tables.

These example OQL statements illustrate the use of the column constraints and the default keyword.

Example 1

```
create database staff;          // creates the staff database
```

The following insert defines the managers table.

```
create table staff.managers
(
        EmployeeID                      int   NOT NULL   PRIMARY KEY,
        Name                            text  NOT NULL,
        Department                      text  default "Sales",
        Gender                          text,
        Age                             int,
        unique ( EmployeeID )         // indicates that the data in the
                                      // EmployeeID column must be unique.
);
```

For the managers table:

- The EmployeeID and Name columns cannot be NULL.
- The EmployeeID column is the primary key and must be unique.
- If no value is inserted into the Department column for a given record it takes the value "Sales".

Example 2

The following insert creates the staff.employees table.

```
create table staff.employees
(
        EmployeeID              int             NOT NULL   PRIMARY KEY,
        Name                    text            NOT NULL,
        Skills                  list type text,
        Gender                  text,
        Age                     int   // There is no comma here because this
                                      // is the last entry.
);
```

For the staff.employees table:

- The EmployeeID and Name columns cannot be NULL.
- The Skills column is a list of text strings.

Example 3

The following insert creates the staff.contractors table.

```
create table staff.contractors
(
        EmployeeID             int NOT NULL PRIMARY KEY,
        Name                   text NOT NULL,
        Gender                 text,
        Age                    int,
        ExtraInfo              object type vblist,
        volatile
);
```

For the staff.contractors table:

- The ExtraInfo column contains a list of varbinds.

Inserting data into a table

Use the **insert** keyword to insert data into a table.

Example

The following example shows how to use the **insert** keyword:

```
insert into             database_name.table_name
(
        column
        [ , column ]
        [ , column ]
        [ ... ]
)
values
(
        data
        [ , data ]
        [ , data ]
        [ ... ]
);
```

Each data value is inserted into the corresponding column name, so the number of data values (and the data types) must correspond with the number of column names specified. Although it is not good practice, it is acceptable to omit the list of column names. If you choose to omit the column names, the first value specified after the values keyword is inserted into the first column of the database, the second value into the second column, and so on. Additionally, if you omit the column names, the number of values must correspond to the number of columns in the database, so you must either insert a value into each database column or explicitly specify NULL for columns into which you do not wish to insert a value.

The following topics provide further examples of the **insert** keyword.

Example of inserting data into a database

Use this example to orient yourself when you insert data into your own databases.

The population of the sample databases with data is shown in the following examples, which highlight some of the different valid forms of the insert statement.

The following example specifies all the column names.

```
insert into staff.managers
(
        EmployeeID, Name, Department, Gender, Age
)
values
(
        1, "Matt", "Development", "M", 28
);
```

The following example does not specify column names, but is still a valid insert because a value has been specified for each column of the database.

```
insert into staff.managers
values
(
        2, "Janet", "Customer Services", "F", 27
                // no column names are specified, so the values are inserted
                // into the columns in order.
);
```

The following example specifies no value for the `Department` column, which is populated with the default value instead:

```
insert into staff.managers
(
        EmployeeID, Name, Gender, Age
)
```

```
values
(
     3, "Phil", "M", 25
             // No value for Department has been specified. The column
             // therefore takes the default value "Sales" that was specified
             // when the table was created.
);
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example of an invalid insert

Use this example to troubleshoot problems and errors if any occur when you insert data into your databases.

Since `staff.managers` is unique on the `EmployeeID` column, the following insert is invalid, and therefore rejected, because there is already a record with `EmployeeID=3`.

```
insert into staff.managers
(
        EmployeeID, Name, Department, Gender, Age
)
values
(
        3, "John", "Marketing", "M", 26
);
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example list and object datatypes

Use this example to orient yourself when using list and object datatypes.

The following example is an insert into the `staff.employees` table, where the `Skills` column has been defined as `list type text`.

```
insert into staff.employees
(
        EmployeeID, Name, Skills, Gender, Age
)
values
(
        6, "Matt", [ "HTML", "C++", "Java" ], "M", 24
);
```

The following example is an insert into the `staff.contractors` table, where the `ExtraInfo` column has been defined as `object type vblist`:

```
insert into staff.contractors
(
        EmployeeID, Name, Gender, Age, ExtraInfo
)
values
(
        11, "James", "M", 22, { ContractLength=6, Department="Marketing" }
);
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Selecting data from a table

You can query the data in a table using the **`select`** keyword. Use these examples to help you use the **`select`** keyword.

Syntax

The following syntax shows how to use the **`select`** keyword to retrieve data from a table.

```
select                  comma_separated_column_list_or_wildcard
from                    database_name.table_name
[ where                 conditional_test ]
[ order by              field_name [asc|desc] ];
```

The * symbol can be used as a wildcard in a select statement to return all the columns of the table. Alternatively a comma-separated list of columns can be specified.

If you specify an **`order by`** clause, then results are returned in ascending order by default. NULL values are returned first when the results are in ascending order. Ordering of results in descending order is the exact opposite of the ordering of results in ascending order.

Example 1

The following example shows how to use the **`select`** statement within the OQL Service Provider to query the `staff.managers` table (the following example output is abbreviated).

```
|phoenix:1.> select * from staff.managers;
|phoenix:2.> go
.....
{
        EmployeeID=1;
        Name='Matt';
        Department='Development';
        Gender='M';
        Age=28;
}
{
        EmployeeID=2;
....
....
}
( 5 record(s) : Transaction complete )
```

Example 2

The following example shows a **`select`** statement that retrieves only specific fields from the `staff.managers` table.

```
|phoenix:1.> select Name, Gender from staff.managers;
|phoenix:2.> go
.....
{
        Name='Matt';
        Gender='M';
}
{
        Name='Irene';
        Gender='F';
```

```
}
{
        Name='Ernie';
....
....
}
( 5 record(s) : Transaction complete )
```

Example 3

The following example uses a where clause to restrict the results.

```
|phoenix:1.> select EmployeeID, Name from staff.managers
|phoenix:2.> where Department = "Marketing";
|phoenix:3.> go
.
{
        EmployeeID=4;
        Name='John';
}
( 1 record(s) : Transaction complete )
```

Example 4

The following example shows how to use a **`select DISTINCT`** keyword to retrieve a single row for each type of data; for example a single row for each department.

```
|phoenix:1.> select DISTINCT Department from staff.managers
|phoenix:2.> go
.
{
        Department='Development';
}
{
        Department='Marketing';
}
{
        Department='Sales';
}
( 3 record(s) : Transaction complete )
```

Counting rows in a table

You can count the number of rows in a table using the **`select`** keyword.

Syntax

The following syntax shows how to use the **`select`** keyword to count the number of rows in a table.

```
select                  count(*)
from                    database_name.table_name
[ where                 conditional_test ]
```

Example 1

The following example shows how to use the **`select`** statement within the OQL Service Provider to count the number of rows in the `staff.managers` table.

```
|phoenix:1.> select count(*) from staff.managers;
|phoenix:2.> go
.....
{
        Count=5;
}
( 1 record(s) : Transaction complete )
```

Conditional tests in OQL

Use comparison operators in OQL, for example in a **select where** statement, to perform conditional tests.

The following table describes the comparison operators that you can use.

Table 10. Comparison operators in OQL

Symbol	Description
=	Equal to.
<>	Not equal to.
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
like	Compares for similarity using UNIX-style regular expressions. The like operator is more powerful than the traditional SQL implementations that use the more limited token matching for comparison.
in	Searches for the presence of a record within a specified table (using a subquery). The in operator can also be used to determine whether a specified value is contained in a list field.
is null	Tests whether the specified column is null (that is, has not been assigned a value).
is not null	Tests whether the specified column is not null (that is, has been assigned a value).

Examples of the `like` operator

Use these examples to help you use the **like** operator in search criteria.

- "Example 1" on page 16
- "Example 2" on page 17
- "Example 3" on page 17

Example 1

The following example query identifies the records in the `employees` table that contain the lowercase letter `r` in the `Name` column.

```
|phoenix:1.> select Name from staff.employees
|phoenix:2.> where Name like ".*[r].*";
|phoenix:3.> go
..
{
        Name='Carl';
}
{
        Name='Sarah';
}
( 2 record(s) : Transaction complete )
```

Example 2

The following example query identifies the records in the `employees` table for which the `Name` column begins with an uppercase `C` or upper `R`.

```
|phoenix:1.> select Name from staff.employees
|phoenix:2.> where Name like "^[CR]";
|phoenix:3.> go
..
{
        Name='Carl';
}
{
        Name='Rob';
}
( 2 record(s) : Transaction complete )
```

Example 3

The following example query would return the complete records of all employees listed in the `contractors` table whose name begins with an uppercase letter `J`.

```
|phoenix:1.> select EmployeeID, Name from staff.contractors
|phoenix:2.> where Name like "J";
|phoenix:3.> go
..
{
        EmployeeID=11;
        Name='James';
}
{
        EmployeeID=13;
        Name='Jane';
}
( 2 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example use of the and Operator

Use this example to help you use the conjunctive operator, and combine two search conditions.

The following example shows how to use the and operator to combine two search conditions.

```
|phoenix:1.> select Name, Gender from staff.managers
|phoenix:2.> where Gender <> "M" and Gender <> "Male";
|phoenix:3.> go
.
{
        Name='Janet';
        Gender='F';
}
( 1 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example use of the or operator

Use this example to help you use the or operator to combine two search conditions.

The following example shows how to use the or operator to combine two search conditions.

```
|phoenix:1.> select Name, Age from staff.employees
|phoenix:2.> where Name="Carl" or Name="Matt";
|phoenix:3.> go
..
{
        Name='Matt';
        Age=24;
}
{
        Name='Carl';
        Age=28;
}
( 2 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example selection based on part of an object

Use this example to orient yourself when retrieving records from tables based on criteria that apply to parts of a table row.

The following example retrieves records from the `staff.contractors` table where `Department="Operations"` within the `ExtraInfo` column.

```
|phoenix:1.> select Name, Age from staff.employees
|phoenix:2.> where ExtraInfo->Department="Operations";
|phoenix:3.> go
..
{
        Name='Richard';
        Age=23;
}
{
        Name='Glenn';
        Age=28;
}
( 2 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Use of `select` to perform subqueries

Subqueries are queries that are embedded within queries using double brackets [[]]. Any valid query can be embedded within the double brackets.

Example 1

The following example retrieves the `Name` and `Age` columns from any record in the `staff.employees` table whose `Name` also exists in the `Name` column of the `staff.managers` table.

```
|phoenix:1.> select Name, Age from staff.employees
|phoenix:2.> where Name in
```

```
|phoenix:3.> ( ( select Name from staff.managers ) );
|phoenix:4.> go
..
{
        Name='Matt';
        Age=24;
}
{
        Name='Rob';
        Age=23;
}
( 2 record(s) : Transaction complete )
```

Example 2

The following example retrieves the Name and Age columns of the managers table where the value of the staff.managers.Age column matches one of the staff.employees.Age columns and is greater than 25.

```
|phoenix:1.> select Name, Age from staff.managers
|phoenix:2.> where Age in
|phoenix:3.> (
|phoenix:4.>                              (
|phoenix:5.>                                  select Age from staff.employees
|phoenix:6.>                                  where Age > 25
|phoenix:7.>                              )
|phoenix:8.> );
|phoenix:9.> go
.
{
        Name='Matt';
        Age=28;
}
( 1 record(s) : Transaction complete )
```

The query returns only one record. Although two records from the managers table have ages that match the employees table, only one of the matches is also over 25.

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the staff database, which contains three tables: managers, employees, and contractors.

Example of using subqueries to search a list

Use this example to help you use a subquery to search a list for information.

The following example uses a subquery to search a list, the Skills column of the staff.employees table. The name of the field to be searched is enclosed in parentheses.

```
|phoenix:1.> select Name, Skills from staff.employees
|phoenix:2.> where ( "C++" in (Skills) );
|phoenix:3.> go
..
{
        Name='Matt';
        Skills=['HTML','C++','Java'];
}
{
        Name='Sarah';
        Skills=['Java','C++'];
}
( 2 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Examples of selecting based on part of a list

You can use these list operators and a conditional test to select records based on part of a list: all, *****, any. The list operators can only be used at the end of an object definition.

The list operators all and ***** return records where the conditional test is true for all items in a list. A query using the keyword any returns records where the conditional test is true for any of the items in the list.

The list operators can also be used in table joins.

Example 1

The following example retrieves all the records that contain `"C++"` anywhere in the list of `Skills`.

```
|phoenix:1.> select Name, Skills from staff.employees
|phoenix:2.> where Skills (any) = "C++";
|phoenix:3.> go
..
{
        Name='Matt';
        Skills=['HTML','C++','Java'];
}
{
        Name='Sarah';
        Skills=['Java','C++'];
}
( 2 record(s) : Transaction complete )
```

Example 2

The following example returns the records that contain *only* `"Perl"` in the list of `Skills`.

```
|phoenix:1.> select Name, Skills from staff.employees
|phoenix:2.> where Skills (*) = "Perl";
|phoenix:3.> go
.
{
        Name='Carl';
        Skills=['Perl'];
}
( 1 record(s) : Transaction complete )
```

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Examples of using any, *, and all to perform table joins
Use these examples to learn how to use the operators to perform table joins.

Selection of data into another table

The **`select into`** statement retrieves data from one table and inserts it into another. The select into command does not delete the existing record.

Syntax

The following syntax shows how to use the **`select into`** statement.

```
select          column_or_wildcard [ , column ... ]
into            destination_database_name.destination_table_name
```

```
from              source_database_name.source_table_name
[ where conditional_test ] ;
```

The columns selected from the source table are inserted into the destination table in order, regardless of the column names and structure of the destination table. Omitting the optional where condition selects all the records from the source table.

If you use a wildcard, all the columns from the source table are selected and inserted in order into the destination table. Any null columns in the source table are skipped in both the source and destination tables, for example, if the fourth column of the source table is null, the fourth column in the destination table is skipped.

If you specify a list of columns to select from the source table, they are inserted into the destination table in the order in which they are specified (even if the order in which they are specified is not the order in which they exist in the source table).

Example 1

The following example selects the values of the Age and Gender columns of the `staff.managers` table and inserts the values into the first two columns of the `staff.employees` table.

```
select Age, Gender into staff.employees from staff.managers;
```

Example 2

The following example selects EmployeeID and Name from any record in the `staff.employees` table for which Name="Carl" and inserts the values into the first two columns of the `staff.managers` table.

```
|phoenix:1.> select EmployeeID, Name
|phoenix:2.> into staff.managers from staff.employees
|phoenix:3.> where Name="Carl";
|phoenix:4.> go
```

Examples of using any, *, and all to perform table joins

Use these examples to learn how to use the operators to perform table joins.

Matching every entry in multiple lists

The following example returns only records where every entry in `db.table1.m_List` is equal to every entry in `db.table2.m_OtherList`. For example, if `m_List = [ 1, 1, 1 ]` and `m_OtherList = [ 1, 1 ]` they match, but if `m_List = [ 1, 2, 1]` and `m_OtherList = [1, 1]`, they do not match.

```
|phoenix:1.> select * from db.table1 , db.table2
|phoenix:2.> where (db.table1.m_List( * ) = db.table2.m_OtherList( * );
|phoenix:3.> go
```

Matching any entry in multiple lists

The following example returns any record where any item in `db.table1.m_List` matches any item in `db.table2.m_OtherList`. For example, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 3, 4 ]` they match because 3 is in both lists. However, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 4, 5]` they do not match.

```
|phoenix:1.> select * from db.table1 , db.table2
|phoenix:2.> where (db.table1.m_List( any ) = db.table2.m_OtherList( any );
|phoenix:3.> go
```

Matching all the entries in one list with any entry in another list

The following example returns any record where all the entries in `db.table1.m_List` match any of the entries in `db.table2.m_OtherList`. For example, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 3, 1, 2, 4 ]` they match, because all the entries in `m_List` match an entry in `m_OtherList`. However, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 3, 2, 4 ]` then they do not match because not all the entries in `m_List` have a match in `m_OtherList`.

```
|phoenix:1.> select * from db.table1 , db.table2
|phoenix:2.> where (db.table1.m_List( * ) = db.table2.m_OtherList( any );
|phoenix:3.> go
```

Matching any entries in one list with all the entries in the second list

The following example returns any record where any of the entries in `db.table1.m_List` match all of the entries in `db.table2.m_OtherList`. For example, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 2, 2 ]` they match because one of the entries in `m_List` matches all the entries in `m_OtherList`. However, if `m_List = [ 1, 2, 3 ]` and `m_OtherList = [ 1, 2, 3 ]` they do not match, because there is no entry in `m_List` that is equal to every entry in `m_OtherList`.

```
|phoenix:1.> select * from db.table1 , db.table2
|phoenix:2.> where (db.table1.m_List( any ) = db.table2.m_OtherList( * );
|phoenix:3.> go
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Updates to records in tables

Use the **update** keyword to update an existing record in a table.

Syntax

The following syntax shows how to use the **update** keyword.

```
update                      database_name.table_name
set                     column = value
                    [ , column = value ... ]
[ where conditional_test ] ;
```

If the update statement is used without a where condition, all records are updated.

Example

The following example updates the Age column of the `staff.managers` table for any records where `Name="John"`.

```
|phoenix:1.> update staff.managers
|phoenix:2.> set Age=27
|phoenix:3.> where Name="John";
|phoenix:4.> go
```

The following topics provide additional examples of the **update** statement.

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Examples of updating a list

Use these examples to learn how to use the **update** keyword in a list.

It is possible to update a single item within the list. To do this, reference the item using an integer that indicates its position in the list, where 0 is the first item.

Example 1

The following example updates the `Age` and `Skills` columns of the `staff.employees` table where `Name="Lisa"`. The following example updates a column of datatype `LIST` by updating the entire list.

```
|phoenix:1.> update staff.employees
|phoenix:2.> set Age=26, Skills=["UNIX", "HTML", "C"]
|phoenix:3.> where Name="Lisa";
|phoenix:4.> go
```

Example 2

The following example updates the `Skills` column, modifying any existing list where `"C"` is the third item and changing that item in the list to `"C++"`.

```
|phoenix:1.> update staff.employees
|phoenix:2.> set Skills(2)=["C++"]
|phoenix:3.> where Skills(2)="C";
|phoenix:4.> go
```

Related reference

Conventions and sample databases
To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Example of updating an object

Use this example to learn how to use the **update** keyword in an object.

The following example updates part of an object, referencing the varbind to be updated using the `->` symbol.

```
|phoenix:1.> update staff.contractors
|phoenix:2.> set ExtraInfo->ContractLength=2
|phoenix:3.> where ExtraInfo->ContractLength=1;
|phoenix:4.> go
```

`ContractLength` is updated to 2 in any records where the `ContractLength` within the `ExtraInfo` field was set to 1.

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Database and table listings

Use the **show** keyword to list the databases, columns, or tables or the current service.

Syntax

The following syntax shows how to use the **show** keyword.

```
show databases ;
show tables from database_name ;
show table       database_name.table_name ;
```

Example 1

The following example shows all the databases of the current service.

```
|phoenix:1.> show databases;
|phoenix:2.> go
.
{
        databases = [ 'staff' ]
}
( 1 Record(s) : Transaction complete )
```

Example 2

The following example shows all the tables from the `staff` database.

```
|phoenix:1.> show tables from staff;
|phoenix:2.> go
.
{
        tables = [ 'managers', 'employees', 'contractors' ]
}
( 1 Record(s) : Transaction complete )
```

Example 3

The following example shows the full schema of the `staff.managers` table.

```
|phoenix:1.> show table staff.managers;
|phoenix:2.> go
.....
{
        schema = {
                 EmployeeID = {
                 DataType = 'text';
                 NotNull = 'Y';
                 PrimaryKey = 'Y';
                 Indexed = 'N';
                 Unique = 'Y';
                 }
        Name = {
                 Datatype = 'text';
.....
.....
               };
}
( 5 Record(s) : Transaction complete )
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Deletion of a record from a database table

You can delete a record from a table using the **`delete`** command.

Syntax

The following syntax shows how to use the **`delete`** command.

```
delete from  database_name.table_name
[ where conditional_test ] ;
```

Attention: Although the where condition is optional, omitting it deletes all the records in the table.

Basic example

The following example deletes all records from the `staff.contractors` table where `Name="James"`.

```
|phoenix:1.> delete from staff.contractors
|phoenix:2.> where Name="James";
|phoenix:3.> go
```

Example of deleting part of an object or list

The following example removes records based on part of the contents of an object.

```
|phoenix:1.> delete from staff.contractors           // Delete records where
|phoenix:2.> where ExtraInfo->Department="Marketing";// the Department
|phoenix:3.> go                                 // in ExtraInfo is Marketing.
```

The following example removes records based on part of the contents of a list.

```
|phoenix:1.> delete from staff.employees
|phoenix:2.> where Skills(0)="Perl";    // Delete records where "Perl"
|phoenix:3.> go                         // is the first list item.
```

The following example removes records based on the entire contents of a list.

```
|phoenix:1.> delete from staff.employees
|phoenix:2.> where Skills=["HTML", "C++", "Java"];
|phoenix:3.> go
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

Deletion of a database or table

You can delete a database or table using the **`drop`** command.

Syntax

The following syntax shows how to use the **`drop`** command

```
drop table database_name.table_name ;
drop database database_name ;
```

Example

The following example deletes the entire `staff.managers` table.

```
|phoenix:1.> drop table staff.managers;
|phoenix:2.> go
```

The following example deletes the entire `staff` database.

```
|phoenix:1.> drop database staff;
|phoenix:2.> go
```

Related reference

Conventions and sample databases

To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.

The `eval` statement

The `eval` statement is used to evaluate the value of a variable or a column within a record, and if necessary, convert it into another data type.

In stitchers, `eval` statements are used in combination with OQL to extract values from records in one database and insert those values into another database. The `eval` statement is used to evaluate specified fields within database records and assign those fields to variables or other database columns, if necessary.

Syntax

The evaluation declaration evaluates a given variable or database record and extracts it to a specified data type.

```
eval ( datatype , string_or_database_column )
```

Where

- The first argument supplied to the `eval` statement is the data type into which the second argument is converted. The data type can be any of the OQL data types, such as text, int, list type text, or object type vblist. You can also use the `multibyte` data type to indicate data that might contain multibyte data. The `multibyte` data type is only available for use in eval statements and is not an OQL data type.
- The second argument is the expression that is to be evaluated, which can include:
  - Variable references, that are preceded by a dollar sign. You can reference global variables like `$HOSTNAME`, product-specific-defined variables such as `$BaseEntity`, and all the environment variables of the shell within which the Network Manager processes are running, for example, the installation directory.
  - References to database columns known to the current scope or outside the current scope that are preceded by the appropriate number of ampersands. If the database column refers to an object, an entry within the object can be extracted using the target identifier `->`.
  - A manipulative statement that can be used to modify complex data types (such as lists or objects), concatenate two items together, delete items from a list, and perform other functions.
  - A special expression using the THIS keyword to move in and out of the data containment model.
  - Combinations of multiple evaluation expressions.

Example

The example shows an `eval` statement which uses OQL.

```
m_CompareDb = eval( text, '&m_Creator' )
insert into kernel.activeModel values (eval( text, "$RECORD" ))
delete from kernel.activeModel where (ObjectId=eval ( text, "&ObjectId" ))
```

Scope of the eval statement

The concept of scope is used throughout the stitcher language to refer to database records from within nested programming loops enclosed in curly braces { }.

The database records that are known within any given scope are said to be *local* to that scope. The `eval` statement uses a system of ampersands to refer either to these local records or to those records that are *outside* the current scope. The only restriction on references to records outside of the present scope is that you cannot reference database records known to scopes that are nested *within* the present scope.

Example

The following example shows three nested scopes:

```
{
        Start of the first scope (SCOPE1)
        ..............
        {
            Start of the second scope (SCOPE2)
            ..............
            {
                Start of third scope (SCOPE3)
                ..............
                ..............
                End of third scope (SCOPE3)
            }
            ..............
            End of the second scope (SCOPE2)
        }
        ...........
        End of the first scope (SCOPE1)
}
```

Where:

- An `eval` statement within scope3 that refers to the EntityName column in a local database record would refer to it as &EntityName.
- An `eval` statement within scope3 that refers to the EntityName column in a record held in scope2 would refer to it as &&EntityName.
- An `eval` statement within scope3 that refers to the EntityName column in a record held in scope1 would refer to it as &&&EntityName.
- It would not be possible to refer to any record held in scope3 from scope2, but scope3 could refer to any record held in scope1 or scope2 using the appropriate number of ampersands.

Quotation marks in eval statements

You must use matching quotation marks in `eval` statements to enclose the variable or database column that you want to be evaluated.

If the `eval` statement is embedded within a set of quotation marks (in either OQL or the stitcher language), the quotation marks enclosing the variable or database column must be of the alternate type to the quotation marks in which the `eval` statement is enclosed. For example, if the `eval` statement is enclosed in single quotation marks (' ') then the variable or database column must be enclosed in double quotation marks (" "'). By default, when the `eval` statement does not occur within an embedded fragment, double quotation marks (" ") are used to enclose the variable or database column to be evaluated.

Single straight back quotes

You must use single straight back quotes (') in `eval` statements to enclose values that are not to be evaluated.

For example, this type of quote is used with the CAT keyword to enclose text strings that are to be concatenated but not evaluated. In the following example, the strings `[ [` and `] ]` are included in the concatenated output but not evaluated.

```
eval( text, 'CAT( &m_Name,'[ [ ',&m_LocalNbr->m_IfIndex,' ] ]' )' )
```

Related reference

Use of the CAT keyword to concatenate lists

Use the **CAT** keyword to concatenate data in an eval statement.

Examples of the eval statement

Use these examples to learn how to use the eval statement.

Example 1

The following example evaluates the contents of the `m_Creator` database column (held one level of scope away from the current scope) and inserts the value into `myVariable`, a previously declared variable of type text.

```
myVariable = eval( text , '&m_Creator' )
```

Example 2

The following example declares an integer variable called `portNum` and assigns to it the value extracted from `m_LocalNbrPort` held within an object called `m_LocalNbr` that is known two levels away from the present scope. `m_LocalNbrPort` is extracted from the object using the target identifier.

```
int portNum = eval( int, "&&m_LocalNbr->m_LocalNbrPort" )
```

Example 3

The following example shows an eval statement in the stitcher language within two `foreach` loops.

```
foreach( connected )
        {
            foreach( uniqueConnector )
            {
                ExecuteOQL
                (
                    "update tempFull.connected
                    set m_RelatedTo = eval(text, '&m_NbrName')
                    where m_Name = eval(text, '&&m_RelatedTo')
                    );"
                );
            }
        }
```

Where:

- Each of the `foreach` loops in the above example specifies a variable of type `RecordList` that has already been assigned the results of an OQL query. The first loop repeats everything within its curly braces for each record in the `RecordList` variable `connected`. Nested within the `foreach( connected ){ }` loop is a second loop, that repeats everything within its curly braces for each record in the `RecordList` variable `uniqueConnector`.
- The `ExecuteOQL` statement that is nested inside both loops updates the `tempFull.connected` database using an eval statement to extract columns from the records held in the two variables:

- The statement `eval(text,'&m_NbrName')` refers to the `m_NbrName` column contained within the local scope, that is, held in the `uniqueConnector` variable.
- The statement `eval(text,'&&m_RelatedTo')` refers to the `m_RelatedTo` column contained one level away from the local scope, that is, held in the `connected` variable.

Character escape sequences

You must escape characters that would otherwise have special meaning in an eval statement using a double-backslash (\\).

The following example shows how to escape the comma, dollar, ampersand, and opening and closing brackets.

```
\\,            -- Escapes the comma.
\\$            -- Escapes the dollar.
\\&            -- Escapes the ampersand.
\\(            -- Escapes the opening parenthesis.
\\)            -- Escapes the closing parenthesis.
```

Multibyte data type

The `multibyte` data type is used to flag data that might be in multibyte form.

Using the multibyte data type in an eval statement

The following example can be used where m_SysLocation might contain multibyte data.

```
eval(multibyte, '&m-ExtraInfo->m_SysLocation')
```

Using the multibyte data type in a configuration file

The following example can be used in a configuration file such as `ModelNcimDb.cfg`.

```
"eval(multibyte, 'LOOKUP(`m_DisplayLabel`, &ExtraInfo, &EntityName)')",
```

Eval statement keywords

Use keywords to perform complex operations within eval statements.

The following table describes the keywords that you can use to perform operations.

Table 11. Table of valid eval statement keywords

Keyword	Synopsis of Keyword Function
APPEND	Appends data to a list or object.
APPENDUNIQUE	Appends data to a list or object only if it does not already exist within the list.
CAT	Concatenates a series of items together.
DELETE	Deletes an item from a list.
FIRSTVALID	Processes a list of possible answers and takes the first valid answer; that is, the first answer that is not null.
LENGTH	Indicates the length of a list or the number of items it holds.
LOOKUP	Allows the use of lookup tables and enables the use of a default human-readable return value in the event of a lookup failure.
IPTOLONG	Converts an IPv4 address into a 32 bit integer.
LONGTOIP	Converts a 32 bit integer into an IPv4 address.

Table 11. Table of valid eval statement keywords (continued)	
Keyword	**Synopsis of Keyword Function**
REGEXPMATCH	Return the value of the first set of parenthesis in a expression.
TIMESTAMP	Generates a human readable timestamp from a 32 bit integer.

The following topics describe how to use the eval statement keywords.

Use of the CAT keyword to concatenate lists

Use the **CAT** keyword to concatenate data in an eval statement.

Syntax

The following syntax shows how to use the **CAT** keyword.

```
CAT( comma separated list of items to be concatenated )
```

Any of the following items can be concatenated together:

- A reference to a database record (preceded by the appropriate number of ampersands).
- A reference to a system or Network Manager variable.
- A text string enclosed within single straight back quotes, for example, `‘item‘`.

Example 1

The following example shows how to format the list of items to be concatenated:

```
eval( text, 'CAT(‘UNCONNECTED_NODES / ‘,&m_Subnet,‘ / ‘,&m_SubnetMask)' )
```

If `m_Subnet='172.16.2.0'` and `m_SubnetMask='255.255.255.0'`, the above syntax would evaluate to the following text string:

```
UNCONNECTED_NODES / 172.16.2.0 / 255.255.255.0
```

Example 2

In the following example, the target identifier extracts the value of a varbind within an object.

```
eval( text, 'CAT( &m_Name,‘[ ‘,&m_LocalNbr->m_IfIndex,‘ ]‘ )' )
```

If the value specified for m_Name is `172.16.1.239` and the value specified for m_IfIndex is 63, the result is the following string.

```
172.16.1.239[ 63 ]
```

Use of the DELETE statement to delete data from a list

The DELETE statement removes items from a list.

Syntax

The following syntax shows how to use the **DELETE** statement.

```
DELETE(       List or reference to a column of type list ,
       List or reference to a column of type list           )
```

Example

In the following example, the output would be the result of removing all items in the *MyEmployees* list from the *AllEmployees* list. The *MyEmployees* list is a subset of the *AllEmployees* list.

```
eval( list type text, 'DELETE( &AllEmployees , &MyEmployees )'
```

Use of the FIRSTVALID keyword to process a list of possible answers

Use the **FIRSTVALID** keyword to process a list of possible answers.

Syntax

The following syntax shows how to use the **FIRSTVALID** keyword.

```
FIRSTVALID( comma separated list of possible answers )
```

Example

The following example shows how to process a list of possible answers and retrieve the first non-null value as an answer.

Note: You can nest the LOOKUP eval keyword within the FIRSTVALID keyword, as shown in the following example.

```
model = eval( text, ''FIRSTVALID(&m_ExtraInfo->physicalChassis-
>model ,&m_ExtraInfo->m_ModelName, &m_ExtraInfo->m_EntPhysModelName,
LOOKUP( &m_ExtraInfo->m_EntVendorType, &&entPhysicalVendorType) )')
```

Assign a value for the model field within the DNCIM physicalChassis table by looking for the first non-null value from the following items in the record. Process these in order and use the first non null value that is encountered:

- `m_ExtraInfo->physicalChassis->model`
- `m_ExtraInfo->m_ModelName`
- `m_ExtraInfo->m_EntPhysModelName`
- `m_ExtraInfo->m_EntVendorType`, using the ncim enumerations to find the vendor mapped to.

Use of the LENGTH keyword to find the number of items in a list

The **LENGTH** keyword returns the number of items in the specified list.

The following example would return the number of items in the*AllEmployees* list.

```
eval( int, 'LENGTH( &AllEmployees )'
```

Use of the LOOKUP keyword to enable lookup tables

The **LOOKUP** keyword allows the use of lookup tables and enables the use of a default human-readable return value in the event of a lookup failure.

Example 1

The following example describes a lookup operation on the `ifAdminStatus` MIB variable, in which `ifAdminStatus` strings are mapped to their enumerated values.

```
{
        ifAdminStatus =
        {
                up = 1,
                down = 2,
```

```
                testing = 3
        }
}
```

Example 2

The following `eval` clause performs a lookup of the enumerated value of the string up within this record and returns a value of 1. The default return value in the event of a lookup failure is `NULL`.

```
eval( text, 'LOOKUP( 'up',&ifAdminStatus )'
```

Example 3

In this clause, no default human-readable return value in the event of a lookup failure has been provided. This is therefore equivalent to the `eval` clause shown below:

```
eval( text, ( '&ifAdminStatus->up' )
```

Example 4

The following `eval` clause performs a lookup of the enumerated value of the string dummy within this record and provides the option of a default human-readable return value. The string dummy does not exist in the record and therefore this clause returns the defined human-readable return value of unknown.

```
eval( text, 'LOOKUP( 'dummy',&ifAdminStatus, 'unknown')'
```

Use of the IPTOLONG keyword to convert between IPv4 addresses and integers

The IPTOLONG keyword provides a mechanism to convert an IPv4 address into a 32 bit integer.

Example

The following example converts the `1.2.3.4` IP Address into number `16909060`.

```
int ipNumber = eval(int, 'IPTOLONG(`1.2.3.4`)')
```

To convert in the opposite direction, you need the LONGTOIP keyword. The LONGTOIP keyword provides a mechanism to convert a 32 bit integer into an IPv4 address.

The following example would convert the number `16909060` into IP Address `1.2.3.4`.

```
text ipAddress = eval(text, 'LONGTOIP(16909060)')
```

Use of the LONGTOIP keyword

The LONGTOIP keyword provides a mechanism to convert a 32 bit integer into an IPv4 address.

Example

The following example converts the number `16909060` into IP Address `1.2.3.4`.

```
text ipAddress = eval(text, 'LONGTOIP(16909060)')
```

Use of the REGEXPMATCH keyword

The REGEXPMATCH keyword provides a mechanism to perform regular expression matching in order to extract string or numerical data from variables.

Example

The following example retrieves an interface entry value from the variable `&LocalPriObj`.

```
int ifEntry1 = eval(int,'REGEXPMATCH(&LocalPriObj, `^ifEntry\.(\d+)$`)')
```

If the variable `&LocalPriObj` contains the value `ifEntry99`, then the above REGEXPMATCH operation returns the value 99.

Use of the TIMESTAMP keyword

The TIMESTAMP keyword generates a human-readable timestamp from a 32 bit integer containing the UNIX time.

Example

The following example would display the current time in the format YYYY-MM-DD HH:MM:SS (for example, 2007-05-24 15:45:19).

```
text timeStamp = eval(text,'TIMESTAMP($TIME, `%a %b %d %H:%M:%S %Y`)')
```

This timestamp is the result of the example: Thursday, May 24 15:45:19 2007

Chapter 2. Stitchers and stitcher language

Stitchers are pieces of code that are used by different Network Manager processes. They take information from one database, process it, and place the information in its new form in another database, or send the information to another process.

Stitchers are used by the following Network Manager processes:

Discovery, ncp_disco
The Discovery engine uses stitchers to move information between databases and to build network topology. You can change or add new stitchers to meet custom discovery requirements; for example, to add and remove devices, or to configure nonstandard configurations, such as non-standard naming of interfaces.Discovery stitchers are stored in the following locations:

- Text-based discovery stitchers (text files with a .stch extension): `$NCHOME/precision/disco/stitchers/`
- Precompiled discovery stitchers : `$NCHOME/precision/platform/`*`platform`*`/lib/`, where *platform* is the operating system on which Network Manager is running.
- dNCIM stitchers: `$NCHOME/precision/disco/stitchers/DNCIM`

For more information on the discovery stitchers, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Event Gateway, ncp_g_event
The Event Gateway uses stitchers to match events to an entity, perform a topology lookup, and then use the topology data retrieved to enrich the event data..

Event Gateway stitchers are stored in the following location: `$NCHOME/precision/eventGateway/stitchers/`.

For more information on the Event Gateway stitchers, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Root-cause analysis plug-in to the Event Gateway
The RCA plug-in uses stitchers to perform root-cause analysis on events passed from the Event Gateway.

RCA plug-in stitchers are stored in the following location: `$NCHOME/precision/eventGateway/stitchers/RCA`.

For more information on the RCA stitchers, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Related reference
Discovery stitchers
Stitchers are processes that transfer, manipulate, and distribute data between databases. The discovery stitchers also process the information collected by the agents and using this information to create the network topology.

Stitcher formats

Stitchers have two formats, *text-based* or *precompiled*.

Text-based stitchers
Text-based stitchers are defined in a plain text file using the stitcher language. The text file defines the processing that the stitcher performs and the reasons why the stitcher is triggered. A text-based stitcher can invoke any other stitcher regardless of whether it is precompiled or text-based

Precompiled stitchers

Precompiled stitchers are written in C++ and are shipped with Network Manager. The precompiled stitchers are designed for computationally intensive operations that they can handle more efficiently than the text-based stitchers.

The precompiled stitchers are controlled by a text-based stitcher, which contains the stitcher trigger conditions. When the text-based stitcher is executed, it calls and executes the precompiled stitcher.

Stitcher structure

Each stitcher consists of two main sections, the *stitcher triggers* and *stitcher rules*.

The following topics describe each of the sections of a stitcher.

Stitcher triggers

A stitcher trigger specifies the actions or conditions that cause the stitcher to run.

Stitchers can be triggered by any of the following conditions:

- The completion of other processes (for example, finders, agents, and other stitchers).
- The insertion of data into a specified database table.
- The end of a specified discovery cycle phase.

Stitchers can also act:

- According to a preconfigured time frequency.
- When asked to do so, that is, on-demand.

You can retrieve stitcher trigger information for the discovery stitchers by performing an OQL query on the stitchers.triggers table in DISCO. This table is created and updated by a periodic scan that DISCO performs on the stitcher files in the `$NCHOME/precision/disco/stitchers` directory.

You can run a specific stitcher by entering the stitcher name in the only field of the `stitchers.actions` table, whereupon DISCO will attempt to execute the specified stitcher. For example, if you update the `stitchers.actions` table value with 'Restitcher', then DISCO will attempt to restitch the topology. You can use this functionality to propagate any modifications throughout the topology.

Stitcher rules

Stitcher rules, written in the stitcher language, define the processing that the stitcher conducts. Stitcher language commands allow the stitchers to make OQL statements to retrieve information from a database and manipulate it. The end result is the distribution of processed information into the appropriate final destination table.

Stitcher language

The stitcher language is used to write the stitcher rules. The rules of a stitcher are written in a stitcher text file; each stitcher has a text file. The stitcher text file also contains the stitcher trigger conditions.

The following topics describe the stitcher language.

Stitcher text file structure

All discovery stitchers have a text file associated with them. The stitcher text files consist of a series of structural statements that enclose the stitcher rules (the actual processing) and the trigger conditions.

The discovery stitcher text files are located under `$NCHOME/precision/disco/stitchers`. The structure of a stitcher text file depends on the type of stitcher (that is, compiled or text-based discovery stitcher).

The following topics describe the text file structure of precompiled and text-based discovery stitchers.

Related reference

Stitcher rules

The stitcher rules are specified within the `StitcherRules{}` section of a stitcher. Stitcher rules determine how a stitcher functions.

Structure of precompiled stitchers

The precompiled stitchers are identified as such in the text file with the `CompiledStitcher{}` statement, which encloses the trigger conditions.

The following syntax shows the full structure of a precompiled stitcher.

```
!CompiledStitcher
{
        StitcherTrigger
        {
            List of stitcher trigger conditions
        }
}
```

Structure of text-based stitchers

Text-based discovery stitchers are identified as such in the text file with the `UserDefinedStitcher{}` statement, which encloses the trigger conditions and the stitcher rules. Stitcher rules carry out the actual stitcher processing.

The following syntax shows the full structure of a text-based discovery stitcher.

```
UserDefinedStitcher
{
        StitcherTrigger
        {
            List of stitcher trigger conditions
        }
        StitcherRules
        {
            List of stitcher rules
        }
}
```

Stitcher trigger conditions

The stitcher trigger conditions are specified within the `StitcherTrigger{ }` section of a stitcher. They specify the conditions that must be met for the stitcher to be executed. 'Triggers' can be used to set off stitchers based on an occurrence. The type of trigger defines which occurrence activates the trigger.

It is possible to enclose multiple stitcher triggers within the `StitcherTrigger{}` section, each of which must be terminated by a semicolon.

Table 12 on page 37 lists the triggers that can be specified.

Table 12. Triggers and trigger types

Trigger name	Trigger type
ActOnDemand	On Demand
ActOnEvent	On Event
ActOnTableDelete	On Table Action
ActOnTableInsert	On Table Action
ActOnTableUpdate	On Table Action
ActOnTimedTrigger	On Timer
DiscoRequiresAgents	On Completion

Table 12. Triggers and trigger types (continued)	
Trigger name	**Trigger type**
DiscoRequiresLastPhase	On Completion
DiscoRequiresPhase	On Completion
RequiresStitchers	On Completion

ActOnDemand();

Use the `ActOnDemand();` stitcher trigger condition to have the stitcher start when it is invoked by the user, or by another stitcher.

No input is required. The following syntax shows how to use the `ActOnDemand();` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnDemand();
}
```

ActOnEvent()

Use the `ActOnEvent();` stitcher trigger condition to start the stitcher when an event any of the specified event states is received from Tivoli Netcool/OMNIbus.

Syntax

The following syntax shows how to use the `ActOnEvent();` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnEvent( ( "event_state", "event_state" );
}
```

Example of event states include Deleted, Occurred, and Reawakened. For a complete list of event states, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

ActOnTableDelete()

Use the `ActOnTableDelete( );` stitcher trigger condition to specify that the stitcher starts every time a row is deleted from the specified database table.

The following syntax shows how to use the `ActOnTableDelete( );` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnTableDelete( "database_name", "table_name" );
}
```

ActOnTableInsert( );

Use the `ActOnTableInsert( );` stitcher trigger condition to specify that the stitcher starts every time data is inserted into the specified database table.

The following syntax shows how to use the `ActOnTableInsert( );` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnTableInsert( "database_name", "table_name" );
}
```

ActOnTableUpdate()

Use the `ActOnTableUpdate( );` stitcher trigger condition to specify that the stitcher starts every time a row is updated in the specified database table.

The following syntax shows how to use the `ActOnTableUpdate( );` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnTableUpdate( "database_name", "table_name" );
}
```

ActOnTimedTrigger();

Use the `ActOnTimedTrigger();` stitcher trigger condition to start the stitcher based on a frequency, which is evaluated relative to the time DISCO started.

Syntax

The following syntax shows how to use the `ActOnTimedTrigger();` stitcher trigger condition.

```
StitcherTrigger
{
    ActOnTimedTrigger( ( frequency_attribute ) values ( value ); );
}
```

Frequency attributes

The following table lists the frequency attributes of the `ActOnTimedTrigger();` stitcher trigger condition.

Table 13. Frequency attributes of the `ActOnTimedTrigger();` stitcher trigger condition

Attribute	Description	Value passed to attribute
`m_DayOfWeek`	Specifies that the stitcher is to be executed on a certain day every week.	Integer representing the day, where Sunday equals `0` and Saturday equals `6`.
`m_DayOfMonth`	Specifies a specific day of the month for the stitcher to run.	Integer from `1` to `31` representing the day of the month on which to run the stitcher. If you specify `31` and the present month only has 28, 29 or 30 days, the timer automatically defaults to the last day of the month.
`m_TimeOfDay`	Specifies a specific time of day for the stitcher to run.	An integer for the time of day in 24 hour format, for example, 2pm is indicated as `1400`.
`m_Interval`	Specifies a specific time interval in hours during which the stitcher runs.	An integer representing the hours during the day when the stitcher runs. **Note:** This value also determines the time that elapses before the stitcher is first run.
`m_IntervalSeconds`	Specifies a specific time interval in seconds during which the stitcher runs.	An integer representing the interval in seconds when the stitcher runs. **Note:** This value also determines the time that elapses before the stitcher is first run.

It is possible to combine either the weekly (`m_DayOfWeek`) or monthly (`m_DayOfMonth`) time interval options with the time of day (`m_TimeOfDay`) option to specify exactly when you want the stitcher to

execute. However, it is not possible to combine any of the configuration options with the `m_Interval` or `m_IntervalSeconds` options, which take precedence over any other attribute.

DiscoRequiresAgents();

Use the `DiscoRequiresAgents();` stitcher trigger condition if you want the stitcher to start only when operations for the specified discovery agents are completed.

The following syntax shows how to use the `DiscoRequiresAgents();` stitcher trigger condition. The name of each agent must be enclosed in double quotes. If multiple agents are specified the list must be separated by commas.

```
StitcherTrigger
{
    DiscoRequiresAgents( "Discovery_agent" );
}
```

DiscoRequiresLastPhase();

Use the `DiscoRequiresLastPhase();` stitcher trigger condition to have the stitcher start only on completion of the last discovery cycle phase.

The following syntax shows how to use the `DiscoRequiresLastPhase();` stitcher trigger condition. No input is required.

```
StitcherTrigger
{
    DiscoRequiresLastPhase();
}
```

DiscoRequiresPhase();

Use the `DiscoRequiresPhase();` stitcher trigger condition to have the stitcher start only on completion of a specified discovery phase.

The following syntax shows how to use the `DiscoRequiresPhase();` stitcher trigger condition.

```
StitcherTrigger
{
    DiscoRequiresPhase( Integer_indicating_discovery_phase );
}
```

RequiresStitchers( );

Use the `RequiresStitchers( );` stitcher trigger condition to have the stitcher start when a specified stitcher or stitchers have completed their operation.

The following syntax shows how to use the `RequiresStitchers( );` stitcher trigger condition. The name of each stitcher must be enclosed in double quotes. If multiple stitchers are specified the list must be separated by commas.

```
StitcherTrigger
{
    RequiresStitcher( "Stitcher" );
}
```

Examples of timed triggers

Use these examples to orient yourself when you configure triggers to run stitchers at predetermined times.

A stitcher can only contain one `ActOnTimedTrigger` trigger, that is, only one trigger activated according to a frequency. So, for example, if you want to set a trigger so that a specific stitcher action is performed

on two days in a week such as Monday and Thursday, you need to define two stitchers and place an `ActOnTimedTrigger` trigger in each.

Example 1

In the following example, the stitcher is configured to run every 60 hours.

```
StitcherTrigger
{
    ActOnTimedTrigger
    (
        ( m_Interval ) values ( 60 );
    );
}
```

Example 2

In the following example, the stitcher is configured to execute every Monday at 3:15 pm.

```
StitcherTrigger
{
    ActOnTimedTrigger
    (
        ( m_DayOfWeek, m_TimeOfDay ) values ( 1, 1515 );
    );
}
```

Stitcher rules

The stitcher rules are specified within the `StitcherRules{}` section of a stitcher. Stitcher rules determine how a stitcher functions.

The following topics describe the processing that takes place within the curly braces of the `StitcherRules{}` section of the text-based stitchers. The stitcher rules are separated by spaces.

Related reference

Stitcher text file structure
All discovery stitchers have a text file associated with them. The stitcher text files consist of a series of structural statements that enclose the stitcher rules (the actual processing) and the trigger conditions.

Stitcher language building blocks
To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

Variables in the stitcher rules

Before you can use variables, you must first declare the variables in the stitcher rules.

The declaration is in the following format.

```
datatype name = initial_value ;
```

You can declare variables anywhere within the `StitcherRules{}`. Although variables must be assigned an initial value, it can be NULL. After a variable has been declared it can be used throughout the current stitcher, but can only accept values of the appropriate datatype. Some examples of variable declaration are shown below.

The following example declares an integer variable called `router` and assigns to it an initial value of `0`.

```
int router = 0;
```

After a variable has been created, its value can be updated at any time. It is also possible to assign the result of an `eval` statement or a calculation to a variable.

Example

The following example declares a text variable called `router2` and assigns to it the string "`upper`".

```
text router2 = "upper";
```

Related reference

Precedence and association of operators
The rules for precedence and association of operators determine the grouping of operators with operands, and indicate the order in which the operators in an expression are executed.

The RecordList and Record datatypes

Variables of the `RecordList` and `Record` datatypes are used to store retrieved OQL records in a variable.

A variable with one of these datatypes is defined, like a text or integer variable, by specifying the datatype and variable name and assigning a value to the variable. The results of an OQL query can be assigned to the variable using the `RetrieveOQL();` rule, which contains an OQL query enclosed in double quotes.

Example of RecordList

The following example declares a variable called `discoRes`, specifies the datatype for the variable as `RecordList`, and assigns the results of a query on the `disco.status` database table to `discoRes`.

```
RecordList discoRes=RetrieveOQL           // Declares discoRes to be a
        (                                 // RecordList variable and
            "select m_DiscoveryMode       // assigns the results of the
            from disco.status;"           // query to discoRes.
        );
```

Example of Record

The `Record` datatype is used in the same way, but variables of the `Record` datatype only hold one record.

The following example shows the `Record` datatype in use.

```
Record newEntity = GetInScopeRecord();
```

Related reference

RetrieveOQL()
The `RetrieveOQL();` stitcher rule executes an OQL query that is expected to return data, against the current process. An optional record can be passed in to be added to the scope of the OQL query.

GetRecordFromScope()
The `GetRecordFromScope();` rule retrieves a record from the specified scope (for example, for use when nested within a `foreach` loop).

Variable declaration and use

Variables of the `Record` and `RecordList` types do not have to be declared on the same line as the assignment of the output from stitcher rules like `RetrieveOQL();`.

As long as the variable has previously been declared it can be assigned the output from a `RetrieveOQL();` rule. The following example is also a valid use of the `Record` datatype.

```
Record newEntity = NULL;
newEntity = GetInScopeRecord();
```

Accessing fields in records

You can access fields within a named record by using the @ character.

Accessing fields within a named record

The @ character allows top-level fields to be set, and can be used to retrieve nested fields.

Note: The @ character does not apply to in-scope records.

Retrieving data from a record

```
Record inScopeCopy = GetInScopeRecord();
```

Retrieving a field

```
int i = @inScopeCopy.integerMemberField;
```

Retrieving a nested field

```
text nested = @inScopeRecordCopy.ExtraInfo.ExampleSubfield;
```

Writing data to a record

The following examples write data to a record:

```
Record newRec;
@newRec.intField = i;
@newRec.anotherIntField = eval(int, '$i');
@newRec.textField = "constant string";
```

The following example extracts the field name from another variable and then writes the data to a record:

```
text myField = "customField";
@newRec.eval(text, '$myField') = "customValue";"
```

Writing nested objects and lists

```
Record newRec;
@newRec.m_ExtraInfo->m_NestedObject   = eval(text, '$internetNodeIP');
@newRec.m_ExtraInfo->m_NestedList(2)   = eval(text, '$internetNodeIP');
@newRec.m_ExtraInfo->m_DeeperNest->m_VeryNestedList(0)   =
 eval(text, '$internetNodeIP');
```

The above code builds a record like the following example:

```
        m_ExtraInfo={
                m_NestedObject='99.99.99.99';
                m_NestedList=['','','99.99.99.99'];
                m_DeeperNest={
                        m_VeryNestedList=['99.99.99.99']
                }
        }
```

Looping within the stitcher rules

The stitcher language relational operators can be used to compare variables and perform conditional tests.

You use the relationship operators for variable comparison and conditional tests in conjunction with the stitcher language programming constructs such as `for`, `foreach`, `while` and `if`. These constructs can be nested within each other.

Related reference

Stitcher language building blocks

To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

The for loop

The `for` loop is used to repeat a set of rules a given number of times.

The `for` loop takes the following format.

```
for( variable_assignment ; conditional_test ; change_to_variable )
        {
            List of stitcher rules to execute
        }
```

The process flow of the `for` loop is as follows:

1. On the first loop, the *`variable_assignment`* assigns a value to a previously declared variable.
2. The *`conditional_test`* is evaluated.
3. If the test evaluates true:
   - The stitcher rules within the curly braces are executed.
   - The *`change_to_variable`* is performed.
   - The loop returns to step 2.
4. When the *`conditional_test`* evaluates false, the loop terminates.

Example

The following example shows a `for` loop. This loop repeats until `variable1` is no longer less than `variable2` (`variable1` is increased by 1 on each complete loop).

```
int variable1 = 0;                // Declares variable1 as an integer.

for( variable1 = 0 ; variable1 < variable2 ; variable1 = variable1 + 1 )
        {
            List of stitcher rules to execute
        }
```

Related reference

Stitcher language building blocks

To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

The while Loop

The `while` loop is used to execute a series of instructions while a specified condition remains true.

The `while` loop takes the following format.

```
while( conditional_test )
        {
            List of stitcher rules to execute
        }
```

The `while` loop repeats as long as the conditional test evaluates to true.

The *conditional_test* can contain boolean expressions containing AND and OR.

An example `while` loop is shown below.

```
while( count < numberOfLayers )
        {
            List of stitcher rules to execute
        }
```

The above loop repeats as long as the value of the `count` variable is less than the value of `numberOfLayers`.

Related reference

Stitcher language building blocks

To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

The if statement

The `if` statement performs an action if a particular condition is satisfied.

The `if` statement takes the following format.

```
if( conditional_test )
        {
            List of stitcher rules to execute if the condition is TRUE
        }
```

The list of stitcher rules are executed if the conditional test is satisfied. It is also possible to specify a list of stitcher rules to execute if the condition is *not* satisfied, as shown below.

```
if( conditional_test )
        {
            List of stitcher rules to execute if the condition is TRUE
        }
else
        {
            List of stitcher rules to execute if the condition is FALSE
        }
```

Note that the `if` statement also supports the following:

- The `if` statement also supports `else if` clauses.
- The *conditional_test* can contain boolean expressions containing AND and OR.

Example

The following example shows the `if` statement in use. If `myVariable` is equal to 1, the first OQL statement is executed, otherwise the second OQL statement is executed.

```
if( myvariable == 1 )
        {
            ExecuteOQL
                (
                    "insert into database.table
                    (             m_Name,
                                  m_BaseName   )
                    values
                    (            "Agent",
                    (            "BaseName"   );"
                );
        }
else
        {
            ExecuteOQL
                (
                    "insert into another.table
                    (             m_Name,
                                  m_BaseName   )
                    values
                    (            "Agent",           "BaseName"   );"
                );
        }
```

Related reference

Stitcher language building blocks

To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

The foreach Loop

The `foreach` loop performs an action on every record stored in a variable of type `RecordList`.

The `foreach` loop has the following syntax.

```
foreach( variable of type RecordList )
        {
            List of stitcher rules to execute
        }
```

Example

The following example shows how the `foreach` loop repeats the list of stitcher rules for every record in the list.

```
foreach( remoteNames )
        {
            ExecuteOQL
                (
                    "insert into CDPLayer.entityByNeighbor
                        (
                            m_Name, m_NbrName, m_NbrType
                        )
                        values
                        (
                            eval(text, '&m_LocalName'),
                            eval(text, '&m_Name'),
                            eval(int, '$RemoteNeighbor')
                        );"
                );
        }
```

The example assumes that a variable called `remoteNames` has already been defined and a list of OQL records has been assigned to it. The `foreach` loop executes the specified OQL insert for every record in the list, extracting the values to insert from the records using `eval` statements.

Related reference

Stitcher language building blocks
To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

List of stitcher rules

Use this list of stitcher rules for reference when you define the rules that you require.

The concept of scope is important in the stitcher rules. The ampersand (&) syntax element is used to retrieve data from the current in-scope record. This scope can be nested; use multiple ampersansds, for example `&&`, to access nested scopes.

General stitcher rules

Use these stitcher rules for reference when you define generic rules for any type of stitcher.

CommitSQLTransaction()
The `CommitSQLTransaction();` rule commits a transaction in a specified database.

Syntax

The `CommitSQLTransaction();` statement uses following syntax.

```
CommitSQLTransaction( database identifier );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 14. Arguments of CommitSQLTransaction()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`database identifier`	Specifies the database on which the commit transaction is to be performed; for example, DNCIM, NCIM, NCMONITOR.	No	No	No

Example

The following example shows how the rule is used.

```
StartSQLTransaction( "DNCIM" );
        text entityName = NULL;
        foreach(entityNames)
        {
            entityName = eval(text,'&m_Name');
            ExecuteStitcher('PopulateDNCIMObject', entityName, domainId,
dynamicDiscoNode);
        }
        delete(entityNames);
CommitSQLTransaction( "DNCIM" );
```

Related reference

StartSQLTransaction()
The `CommitSQLTransaction();` rule starts a commit transaction in a specified database.

RollbackSQLTransaction()
The `RollbackSQLTransaction();` rule rolls back a transaction in a specified database.

delete()
The `delete` rule removes lists and records that have been created and are no longer needed.

Syntax

The following syntax shows how to use the `delete` rule.

```
delete( variable of type Record or RecordList );
```

Tip: Delete lists after they are no longer needed to release memory.

Example

Some examples are shown below.

```
delete( notIpSwitchNbrs );
delete( ethernetSwitches );
```

Related reference

Stitcher language building blocks
To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

ExecEvalOnRecord()
The `ExecEvalOnRecord();` rule executes the supplied `eval` statement on the named record.

Syntax

The `ExecEvalOnRecord();` statement uses following syntax.

```
ExecEvalOnRecord(record, eval clause  );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 15. Arguments of ExecEvalOnRecord()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`record`	Typically retrieved from a database or passed in to the stitcher as an argument.	No	Yes	No
`eval clause`	Defines the type and name of the field to extract.	No	No	Yes

Example

The following example shows how a text variable called `name` is assigned the evaluation of the `EntityName` field of the `currentRec` variable.

```
text name=ExecEvalOnRecord( currentRec , eval(text, '&EntityName' );
```

ExecuteOQL()

The `ExecuteOQL();` rule tells the stitcher to execute an OQL statement on the databases of the current service. For example, if this rule is run from a discovery stitcher, then the current service is the Discovery engine, ncp_disco.

Syntax

The `ExecuteOQL();` statement uses following syntax.

```
ExecuteOQL( oql string, [optional record]  );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 16. Arguments of ExecuteOQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`oql string`	The OQL command to execute. This can reference members of the optional record, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	Yes	Only within the OQL string.
`optional record`	An optional record that can be passed in to be added to the scope of the OQL query.	No	Yes	No

Example

The following example shows how the `ExecuteOQL()` rule is used to delete all the records in the `finders.pending` table.

```
text oqlDelete = "delete * from finders.pending;";
ExecuteOQL( oqlDelete );
```

In the following example, the optional record is added to the symbol table used for the OQL statement, and can be accessed within the OQL statement using a single ampersand. Here the record `myRecord` contains the field `m_TableName`.

```
text oqlDelete = "delete * from finders.pending;";
ExecuteOQL( "delete from eval(int, &m_TableName);", myRecord );
```

ExecuteSQL()
The `ExecuteSQL();` rule executes a prepared SQL query.

Syntax

The `ExecuteSQL();` statement uses following syntax.

```
ExecuteSQL ( sql data, [optional variable list] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 17. Arguments of ExecuteSQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`sql data`	The prepared SQL command to execute. The prepared SQL command is presented as an SQLData variable. **Note:** SQL queries must not be terminated with a semi-colon.	Yes	Yes	Yes
`optional variable list`	An optional comma-separated list of variables to be used if placeholders (marked by a question mark symbol, ?) are used in the `sql data` argument.	Yes	Yes	Yes

Example

The following example shows how the `ExecuteSQL()` rule is used to execute an SQL statement that was prepared earlier.

```
SQLData myData = PrepareSQL(
              "insert into entityNameCache
              (
                            entityName,
                            domainMgrId
              )
              values
              (
                            ?,
                            ?
              );"
              , "NCIM" ,
              eval(text,'$entityName'),
```

```
              eval(int,'$domainId')
              );

ExecuteSQL ( myData );
```

ExecuteStitcher()
The `ExecuteStitcher();` function runs the specified stitcher.

Syntax

After the invoked stitcher has completed, control is passed back to the original stitcher and the remaining stitcher rules are run in sequence. The following syntax shows how to use the `ExecuteStitcher()` function.

```
ExecuteStitcher( 'stitcher name', [optional variable list] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 18. Arguments of ExecuteStitcher()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`stitcher name`	The name of the stitcher to call.	Yes	No	No
`optional variable list`	An optional comma-separated list of variables to be passed to the stitcher.	Yes	Yes	Yes

No arguments passed

The following example runs the `ProcessAffectedRemoteSwitches.stch` stitcher without passing any arguments.

```
ExecuteStitcher( 'ProcessAffectedRemoteSwitches' );
```

Passing two variables

The following example executes the `SwitchFdbToConnections.stch` stitcher and passes two variables to the stitcher.

```
ExecuteStitcher( 'SwitchFdbToConnections' , 'agent1' , 'agent2' );
```

Use of eval statement in the argument list

The following example executes the `IsInScope.stch` stitcher to determine whether an entity is within the scope of discovery, that is, within the scope defined in the scope.zones table. The entity is defined using an IP address and a netmask and the IP address is calculated by using an `eval` statement to evaluate the value of the `$testIp` variable.

```
ExecuteStitcher( 'IsInScope' , eval(text,'$testIp') , netMask, 1 );
```

Returning an argument

The following example shows how the `ExecuteStitcher()` call can return an argument. The called stitcher must use the `SetReturnValue()` rule to return the argument.

In the DetailsRetProcessing.stch stitcher:

```
...
toBeDetected = ExecuteStitcher('DetectionFilter', protocol);
```

In the DetectionFilter.stch stitcher:

```
...
SetReturnValue(toBeDetected);
```

Extraction of variable passed to a stitcher

The stitcher invoked using the `ExecuteStitcher();` rule can extract the variables that have been passed to it by evaluating the special variable `ARG_N` using an `eval` statement, where `N` is the number of the variable (for example, `ARG_1` represents the first variable passed to the stitcher, `ARG_2` represents the second variable, and so on).

ExecuteStitcherOnTimer
The `ExecuteStitcherOnTimer();` function runs the specified stitcher after a delay.

Syntax

When a stitcher calls the another stitcher using the `ExecuteStitcherOnTimer();` function, the first stitcher continues to run. The **ncp_disco** process starts the second stitcher after the specified delay. The second stitcher only uses the arguments that you explicitly pass to it in the function.

The `ExecuteStitcherOnTimer();` function uses the same syntax as the `ExecuteStitcher();` function.

The following syntax shows how to use the `ExecuteStitcherOnTimer()` function.

```
ExecuteStitcherOnTimer( 'stitcher name', [optional variable list], AtTime
( ( m_IntervalSeconds ) values ( seconds ); ); );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 19. Arguments of ExecuteStitcherOnTimer()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`stitcher name`	The name of the stitcher to call.	Yes	No	No
`optional variable list`	An optional comma-separated list of variables to be passed to the stitcher.	Yes	Yes	Yes
`AtTime`	Specifies the number of seconds to wait before calling the stitcher.	Yes	Yes	Yes

Example: Calling a stitcher with a delay

The following example executes the `TestStitcher.stch` stitcher after 12 seconds. The example also passes a single argument to the stitcher, `domain`, which has been defined as `NCOMS`.

```
text domain = "NCOMS";

ExecuteStitcherOnTimer( "TestStitcher", domain )
AtTime
(
```

```
( m_IntervalSeconds ) values ( 12 ) ;
) ;
```

FetchLastInsertedIdForSQL()
The `FetchLastInsertedIdForSQL();` rule retrieves the last inserted unique key caused by an SQL data object. This is useful when you are inserting into a table with automatically incrementing IDs, such as the `entityNameCache` table. When an insert is made the `FetchLastInsertedIdForSQL();` rule can be used to retrieve the automatically incremented id that was applied to that insert. This rule is used together with the `PrepareSQLAutoColumn();` rule.

Syntax

The SQL data object is enclosed in brackets, as shown in the following syntax.

```
intVariable = FetchLastInsertedIdForSQL( SQL data object );
```

Example

The following example shows how the `FetchLastInsertedIdForSQL()` rule is used to retrieve the last inserted unique key caused by an SQL data object.

```
// We need to populate the entityNameCache
// entityId will be auto populated.
SQLData myData = PrepareSQLAutoColumn(
            "insert into entityNameCache
            (
                        entityName,
                        domainMgrId
            )
            values
            (
                        ?,
                        ?
            );"
                        , "NCIM" ,
                        eval(text,'$entityName'),
                        eval(int,'$domainId')
            );

ExecuteSQL(myData);
entityId = FetchLastInsertedIdForSQL( myData );
delete(myData);
```

GetInScopeRecord()
The `GetInScopeRecord();` rule retrieves the record currently in scope. Using this rule is equivalent to running the rule `GetRecordFromScope ()` with a `depth` argument of 0, that is, `GetRecordFromScope (0)`.

The rule requires no input. The results are to be assigned to a variable of type `Record`. The following syntax shows how to use `GetInScopeRecord();`.

```
GetInScopeRecord();
```

Related reference
GetRecordFromScope()

The `GetRecordFromScope();` rule retrieves a record from the specified scope (for example, for use when nested within a `foreach` loop).

GetRecordFromScope()
The `GetRecordFromScope();` rule retrieves a record from the specified scope (for example, for use when nested within a `foreach` loop).

Syntax

The `GetRecordFromScope();` statement uses following syntax.

```
GetRecordFromScope ( depth  );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 20. Arguments of GetRecordFromScope()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
depth	Numbers greater than 0 can be used to extract a record from a deeper nested scope, in the same way that multiple ampersands can be used.	Yes	No	No

Example

Records that are in scope can be accessed using ampersands, as shown in this example.

```
RecordList myList = list;

text currentRecordName = "";

// Upon each iteration through this loop, one record is in scope
foreach (myList)
{
    currentRecordName = eval(text, '&Name')
}
```

The entire record can be retrieved from the same scope using the `GetRecordFromScope()` stitcher rule, as shown in this example.

```
RecordList myList = <whatever>;

Record currentRecord = NULL;

// Upon each iteration through this loop, one record is in scope
foreach (myList)
{
    # depth 0 means the record in the nearest scope
    currentRecord = GetRecordFromScope(0);
}
```

Related reference

The RecordList and Record datatypes
Variables of the `RecordList` and `Record` datatypes are used to store retrieved OQL records in a variable.

GetInScopeRecord()

The `GetInScopeRecord();` rule retrieves the record currently in scope. Using this rule is equivalent to running the rule `GetRecordFromScope ()` with a `depth` argument of 0, that is, `GetRecordFromScope (0)`.

IsInSubnet()
The `IsInSubnet();` rule determines whether a specified IP address is in a given subnet, based on the subnet mask.

Syntax

The `IsInSubnet();` statement uses following syntax.

```
IsInSubnet( ip , subnet , mask );
```

The rule returns an integer where 0 indicates that the IP address is not in the subnet and 1 indicates that the IP address is in the subnet.

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 21. Arguments of ExecuteSQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`ip`	Textual IP address (dot notation for IPv4; colon notation for IPv6).	Yes	Yes	No
`subnet`	Textual subnet (dot notation for IPv4; colon notation for IPv6).	Yes	Yes	No
`mask`	This argument can be specified as a string in the same format as the subnet/IP or as a numerical integer mask.	Yes	Yes	No

Example

The following example shows how the `ExecuteSQL()` rule is used to test whether an IPv4 address passed as an argument is within the scope of a specified Class C subnet.

```
int inScope = 0;
text ip = eval(text, '$ARG_2');
inScope = IsInSubnet( ip, "10.10.10.0", 24 )
```

IsRecordInFilter()
The `IsRecordInFilter()` rule applies the specified filter string to the specified record and returns the following value: 1 if the record passes the filter, 0 if the record does not pass the filter. This technique can be used whenever you need to determine if a record passes a given set of criteria. The technique is used in the discovery process to test whether specific network entities match a given filter entities at various points in the discovery.

Syntax

The `IsRecordInFilter();` statement uses the following syntax.

```
IsRecordInFilter ( filter string, record );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 22. Arguments of IsRecordInFilter()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`filter string`	Filter used to test the record	No	Yes	No
`record`	Record to test	No	Yes	No

Example

The following example shows how the `IsRecordInFilter()` rule is used in the discovery process to check whether the Object ID of a specified Cisco entity adheres to a specified format and whether the entity contains an experimental Cisco IOS.

```
# Filter to determine if a given record is a cisco device with
#    an experimental IOS
#   deviceRec could be anything but for example contains
#
     {           EntityName='bristol-cs55-catos';
                 Address=['','00:02:BA:6D:F3:FF','172.20.72.2'];
                   Description='Cisco Systems WS-C5000 Cisco Catalyst Operating
        System Software, Version 5.2(4)
        Copyright (c) 1995-2000 by Cisco Systems';
                   EntityType=1;
                   EntityOID='.1.3.6.1.4.1.9.5.7';
                   IsActive=1;
                   Status=1;
      )

int filterResult = IsRecordInFilter( 'EntityOID LIKE "^1.3.6.1.4.1.9.*" AND
Description LIKE ".*Experimental Release.*" ' , deviceRec);
```

Log()

The `Log();` rule prints a message to the `.log` log file at the given message level if appropriate to the current message level. This rule also prints to the .trace file if running at debug level 4.

Syntax

The `Log();` statement uses following syntax.

```
Log( level, symbol, [optional symbol list] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 23. Arguments of Log()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`level`	Message level to log the message at. Can be one of the following: • debug • info • warn • error	Yes	Not for the level	No

Table 23. Arguments of Log() (continued)				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`symbol`	Main part of the message to print out.	Yes	Not for the level	No
`[optional symbol list]`	If supplied, these will be appended in order to the string in the output.	Yes	Not for the level	No

Example

The following examples shows how the rule is used.

```
Log("debug", "Subnet Name : ", subnetName);
```

```
Log("warn", "The topology entity for the topology type you are trying to insert into doesn't exist. Connection rejected for topology type: ", topologyType
```

Related reference

Print()

The `Print()` rule sends information to the standard output, but differs from `PrintRecord()` in that the information to be printed must be specified.

MatchPattern()

The `MatchPattern()` rule performs string extraction based on regular expressions. The rule determines how many times a regular expression pattern was found in a target string. The rule also enables you to create variables based on subpatterns found in the target string.

Syntax

The `MatchPattern();` statement uses following syntax.

```
MatchPattern( input string,regular expression );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 24. Arguments of MatchPattern()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`input string`	Search for the given pattern within this string.	Yes	Yes	Yes
`regular expression`	The regular expression pattern to try to match to the input string.	Yes	Yes	Yes

Example 1

The following example shows a use of the `MatchPattern()` rule when both the target string and the pattern string are *defined strings*: The following example returns the value 3 as the pattern was matched three times.

```
int count=MatchPattern( "Hello Hello Hello","Hello" );
```

Example 2

The following example shows a use of the `MatchPattern()` rule when both the target string and the pattern string are *variables*: The following example also returns the value 3.

```
text target="Hello Hello Hello";
text pattern="Hello";
int count=MatchPattern( target,pattern );
```

Example 3

The following example shows an eval statement used to match a pattern against an m_Description field in the in scope record.

```
text pattern='[[:space:]]+OS([^[:space:]]+)[[:space:]]+';
int count=MatchPattern( eval(text,'&m_Description'),pattern);
```

Example 4

The following example shows how to extract an interface index from an input string.

```
# Extract an ifIndex from an input string in a known format
text input = "ifEntry.12";

# The input must start with "ifEntry", followed by a single '.' followed
# by one or more numbers. The numbers are extracted and saved by virtue
# of being within round brackets
text pattern = "^ifEntry\.(\d+)";

# return the number of times the pattern was found in the input string
int patternMatchCount = MatchPattern( input, pattern );

# In this example, we only recognise a match if we matched once on the entire field
# REGEX0 conatins the section of the input that was matched to the whole pattern
if (patternMatchCount == 1 AND REGEX0 == input )
{
        # REGEX1 contains the first match from within round brackets. If multiple
        # parts of the pattern were extracted with multiple pairs of round brackets,
        # they would be stored in symbols REGEX2, REGEX3,... etc
        int ifIndex = eval(int, '$REGEX1');
}
```

String extraction

The `MatchPattern()` rule can also extract text from the target string by creating variables based on sub-patterns found in the target string.

These variables take the name REGEX*N*, where *N* is a number which identifies the variable.

Example

The following example shows how variables are assigned.

```
[1]text target="Testing Testing One 2 Three";
[2]text pattern="([^[:space:]]+)[[:space:]]+([^[:space:]]+)[[:space:]]+
        ([^[:space:]]+)[[:space:]]+([[:digit:]]+)[[:space:]]+([^[:space:]]+).*";
[3]int count=MatchPattern( target,pattern );
[4]while( count > 0 )
[5]        {
[6]            Print("Count ", count);
[7]             Print("Match ", REGEX0);
[8]            Print("SubMatch1 ", REGEX1);
[9]             Print("SubMatch2 ", REGEX2);
[10]              Print("SubMatch3 ", REGEX3);
[11]              Print("SubMatch4 ", REGEX4);
[12]              Print("SubMatch5 ", REGEX5);
[13]
[14]
```

```
[15]            count = count - 1;
        }
```

The round brackets in the pattern string defined in line 2 identify a subpattern to be extracted and placed within a REGEX variable. In this example there is one match only; hence there is one variable (REGEX0), which matches the full pattern, and five sub-matches, (REGEX1 to REGEX5), set as shown in the following table.

Table 25. Extraction of variables based on sub-patterns

Variable	Value	Type
REGEX0	Testing Testing One 2 Three	Match
REGEX1	Testing	Sub-match
REGEX2	Testing	Sub-match
REGEX3	One	Sub-match
REGEX4	2	Sub-match
REGEX5	Three	Sub-match

If more than one match is found, more REGEX variables are created.

For example, if there were three matches, then 18 REGEX variables would be created. In this case, the variables REGEX0, REGEX6, and REGEX12 would correspond to the full matching pattern, while variables REGEX1-5, REGEX7-11, and REGEX13-18 would correspond to the subpatterns within each full match.

Note: If you run the `MatchPattern()` rule a second time, then it overwrites previous values held in the REGEX variables. It is recommended that you store any extracted data prior to running the `MatchPattern()` rule a second time.

Example of pseudowire
Use this realistic example based on data received from an MPLS agent for a real-life example of a pseudowire data string.

Assume that the following pseudowire data string is retrieved from the agent.

```
Layer-2 VPN Statistics:
 Instance: vpls1

 Local interface: fe-1/3/0.0, Index: 71
 Multicast packets:               375747
 Multicast bytes  :             34780885
 Flooded packets  :                96012
 Flooded bytes    :              9213657

 Local interface: vt-1/2/0.32768, Index: 72
 Remote PE: 10.1.230.4
 Multicast packets:               174307
 Multicast bytes  :             14449864
 Flooded packets  :               615013
 Flooded bytes    :             50990719

 Instance: vpls1

 Local interface: fe-1/3/0.0, Index: 71
 Multicast packets:               375747
 Multicast bytes  :             34780885
 Flooded packets  :                96012
 Flooded bytes    :              9213657

 Local interface: vt-1/2/0.32768, Index: 72
 Remote PE: 10.1.230.4
 Multicast packets:               174307
 Multicast bytes  :             14449864
 Flooded packets  :               615013
 Flooded bytes    :             50990719
```

You can extract the data from this string by writing a stitcher which incorporates code with the `MatchPattern()` rule.

```
[1]text pattern=
   "Instance:[[:space:]]+([^[:space:]]+)[[:space:]]*[\n\r][[:space:]]
   *[\n\r][[:space:]]*Localinterface:[[:space:]]+([^,]+),[[:space:]]
   +Index:[[:space:]]+([[:digit:]]+)[[:space:]]*[\n\r][[:space:]]*
   Multicast packets[[:space:]]*:[[:space:]]+[[:digit:]]+[[:space:]]*
   [\n\r][[:space:]]*Multicast bytes[[:space:]]*:[[:space:]]+[[:digit:]]
   +[[:space:]]*[\n\r][[:space:]]*Flooded packets[[:space:]]*:[[:space:]]
   +[[:digit:]]+[[:space:]]*[\n\r][[:space:]]*Flooded bytes[[:space:]]*:
   [[:space:]]+[[:digit:]]+[[:space:]]*[\n\r][[:space:]]*[\n\r]
   [[:space:]]*Local interface:[[:space:]]+([^,]+),[[:space:]]+Index:
   [[:space:]]+([[:digit:]]+)[[:space:]]*[\n\r][[:space:]]*Remote PE:
   [[:space:]]+([^ \t\n\r]+)[[:space:]]*[\n\r].*";
[2]int count = MatchPattern( eval ( text,$target ), pattern);
[3]while(count > 0)
[4]        {
[5]            Print("Count ", count);
[6]            Print("Match ", REGEX0);
[7]            Print("SubMatch 1 ", REGEX1);
[8]            Print("SubMatch 2 ", REGEX2);
[9]            Print("SubMatch 3 ", REGEX3);
[10]           Print("SubMatch 4 ", REGEX4);
[11]           Print("SubMatch 5 ", REGEX5);
[12]           Print("SubMatch 6 ", REGEX6);
[13]
[14]           count = count - 1;
[15]       }
```

In line 2 of this example, `$target` is the pseudowire data string received from the MPLS stitcher.

MergeEntities()
The `MergeEntities();` rule merges two variables of type `Record` into a single record, leaving the original records unchanged. If the same field exists in both records, then the `precedence flag` indicates which will be used.

Syntax

The `MergeEntities();` statement uses following syntax.

```
MergeEntities( lhs record , rhs record , precedence flag );
```

The output of the rule can be assigned to a variable of type `Record`.

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 26. Arguments of ExecuteSQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`lhs record`	One record to merge.	No	Yes	No
`rhs record`	Another record to merge.	No	Yes	No
`precedence flag`	A boolean value. If 0, then `rhs record` takes precedence. If 1, then `lhs record` takes precedence.	No	Yes	No

This examples merges two entity records, giving precedence to the right-hand side (rhs) entity.

```
int oldIsMaster = 0;
Record oldEntity = value1;
Record newEntity = value2;
Record mergedRecord = MergeEntities( newEntity, oldEntity, oldIsMaster );
```

PrepareSQL()
The `PrepareSQL();` rule prepares an SQL statement for execution.

Syntax

The `PrepareSQL();` statement uses the following syntax.

```
PrepareSQL ( sql string, database id, [optional variable list] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 27. Arguments of PrepareSQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`sql data`	The prepared SQL command to execute. The prepared SQL command is presented as an SQLData variable. **Note:** SQL queries must not be terminated with a semi-colon.	Yes	Yes	Yes
`database id`	The database ID, as specified in the DbLogins.cfg file, used to identify the database schema that contains the required table.	Yes	Yes	Yes
`optional variable list`	An optional comma-separated list of variables to be used if placeholders (marked by a question mark symbol, ?) are used in the `sql data` argument.	Yes	Yes	Yes

Example

The following example shows how the `PrepareSQL()` rule is used to prepare an SQL statement for execution.

```
SQLData myData = PrepareSQL(
             "insert into entityNameCache
             (
                          entityName,
                          domainMgrId
             )
             values
             (
                          ?,
                          ?
             );"
             , "NCIM" ,
             eval(text,'$entityName'),
             eval(int,'$domainId')
             );

ExecuteSQL ( myData );
```

PrepareSQLAutoColumn()
The `PrepareSQLAutoColumn();` rule prepares an SQL statement for execution and notifies the system of a column which will be automatically incremented as a result of this operation. This enables the

automatically incremented column to be retrieved later using the `FetchLastInsertedIdForSQL();` rule.

Syntax

The `PrepareSQLAutoColumn();` statement uses the following syntax.

```
PrepareSQLAutoColumn ( sql string, auto-incrementing column name, database id,
[optional variable list] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 28. Arguments of PrepareSQLAutoColumn()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`sql data`	The prepared SQL command to execute. The prepared SQL command is presented as an SQLData variable. This command can contain placeholders (marked by a question mark symbol, ?). **Note:** SQL queries must not be terminated with a semi-colon.	Yes	Yes	Yes
`auto-incrementing column name`	Name of the field in the table that is set to automatically increment upon insert.	Yes	Yes	Yes
`database id`	The database ID, as specified in the DbLogins.cfg file, used to identify the database schema that contains the required table.	Yes	Yes	Yes
`optional variable list`	An optional comma-separated list of variables to be used if placeholders (marked by a question mark symbol, ?) are used in the `sql data` argument.	Yes	Yes	Yes

Example

The following example shows how the `PrepareSQLAutoColumn()` rule is used to prepare an SQL statement for execution. An insert is performed into the entityNameCache table. This insert automatically increments the field entityId, which is then retrieved using the `FetchLastInsertedIdForSQL();` rule.

```
SQLData myData = PrepareSQLAutoColumn(
             "insert into entityNameCache
             (
                          entityName,
                          domainMgrId
             )
             values
             (
                          ?,
                          ?
             );"
             ,"entityId"
             ,"NCIM",
             eval(text,'$entityName'),
```

```
              eval(int,'$domainId')
              );

ExecuteSQL ( myData );
entityId = FetchLastInsertedIdForSQL(myData);
delete(myData);
```

Print()
The `Print()` rule sends information to the standard output, but differs from `PrintRecord()` in that the information to be printed must be specified.

Syntax

The `Print();` statement uses following syntax.

```
Print ( string, [optional list of variables] );
```

At least two comma-separated arguments must be specified within the parentheses of `Print()`. The arguments can be any of the following: a list of strings enclosed in quotation marks (which may be empty), a string that is not enclosed in quotation marks (which may be a symbol definition name or may be NULL), an integer, or an `eval` statement.

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 29. Arguments of Print()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`string`	Main part of the message to print out.	Yes	No	No
`optional list of variables`	If supplied, these variables will be appended to `string` in the output.	Yes	Yes	Yes

Example

An example of the `Print()` rule that prints the current time to the standard output is shown below.

```
Print ( "CPU time:", eval (text, "$TIME") );
```

This example prints out stitcher start time.

```
text timeString = eval(text, $FTIME);
Print("This sticher is starting at time ", timeString);
```

Related reference

Log()
The `Log();` rule prints a message to the `.log` log file at the given message level if appropriate to the current message level. This rule also prints to the .trace file if running at debug level 4.

PrintRecord()
The `PrintRecord();` stitcher rule prints the entire record currently in scope to standard output. This can be used for debugging purposes.

Syntax

The `PrintRecord();` statement uses following syntax.

```
PrintRecord ( [optional string], record );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 30. Arguments of PrintRecord()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`optional string`	An optional message to preappend to the `record` in the output.	Yes	No	No
`record`	The record to print.	No	Yes	No

Example

In the following example, `PrintRecord()` is used to print the records within a variable of datatype `RecordList` called `snmpResults`.

```
foreach( snmpResults )
        {
            PrintRecord(snmpResults);
        }
```

Example

In the following example, `PrintRecord()` is used to print a record with preappended text.

```
Record forOutput = record1;
PrintRecord("My record looks like this: ", forOutput);
```

RaiseEvent()

The `RaiseEvent();` rule uses the standard Network Manager alerts library to raise an event and send it to the Tivoli Netcool/OMNIbus Object Server using the Probe for Tivoli Netcool/OMNIbus, nco_p_ncpmonitor.

Syntax

The `RaiseEvent();` statement uses following syntax.

```
RaiseEvent ( event name,
                  severity,
                  type,
                  entity name,
                  description
                  [, optional record with extra info] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 31. Arguments of RaiseEvent()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`event name`	A string value that will be available in the probe rules as the EventName field, and that by default populates the alerts.status EventId field.	Yes	Yes	Yes
`severity`	An integer value that will be available in the probe rules as the Severity field, and that by default populates the alerts.status Severity field.	Yes	Yes	Yes
`type`	An integer value that is restricted to the following values: • 1 (Problem) • 2 (Resolution) • 13 (Information) This value will be available in the probe rules as the ExtraInfo_EVENTTYPE field, and that by default populates the alerts.status Type field.	Yes	Yes	Yes
`entity name`	A string value that will be available in the probe rules as the EntityName field, and that by default populates the alerts.status Node field.	Yes	Yes	Yes
`description`	A string value that will be available in the probe rules as the Description field, and that by default populates the alerts.status Summary field.	Yes	Yes	Yes
`optional record with extra info`	All fields from this record will be available in the probe rules as part of the ExtraInfo field.	No	Yes	No

Example

The following example shows how the `RaiseEvent()` rule is used to raise an event.

```
int eventSeverity = 1;
int eventType = 13; // Information

text eventDescription = "My big old test event";
text eventName = "CustomEvent";
text entityName = "SomeEntity";

Record extraInfo;
@extraInfo.ALERTGROUP = "ITNM Status";

RaiseEvent( eventName,
       eventSeverity,
       eventType,
       entityName,
```

```
        eventDescription,
        extraInfo );
```

RetrieveOQL()
The `RetrieveOQL();` stitcher rule executes an OQL query that is expected to return data, against the current process. An optional record can be passed in to be added to the scope of the OQL query.

Syntax

The `RetrieveOQL();` statement uses following syntax.

```
RetrieveOQL( oql string [, optional record] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 32. Arguments of ExecuteSQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`oql string`	OQL command to execute. This can reference members of the optional record, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	Yes	Yes
`optional record`	If present, this will be added so that it can be dereferenced in the OQL string.	No	Yes	No

Example

The following example shows the results of a query on the `finders.returns` table being assigned to a previously declared variable of datatype `RecordList` called `devicesFound`.

```
devicesFound = RetrieveOQL( "select m_Name from finders.returns;" );
```

Example

The following example shows the results of a query on the disco.agents table and displays each record retrieved using a `foreach` loop.

```
RecordList results = RetrieveOQL( "select * from disco.agents;" )

foreach (results)
{
    Record current = GetInScopeRecord();
}
```

Related reference

The RecordList and Record datatypes

Variables of the RecordList and Record datatypes are used to store retrieved OQL records in a variable.

RetrieveOQLFromService()
The RetrieveOQLFromService(); rule issues an OQL query on the databases of the specified service. An optional record can be passed in to be added to the scope of the OQL query. The results can be assigned to a variable of the type RecordList.

Syntax

```
RetrieveOQLFromService( oql string, service name [,optional record] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 33. Arguments of RetrieveOQLFromService()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
oql string	OQL command to execute. This can reference members of the optional record argument, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	No	Yes
service name	Name of the process to retrieve results from. Available strings can be found by running the command ncp_oql -options.	Yes	No	No
optional record	If present, this is added so that it can be dereferenced in the oql string argument.	No	Yes	No

Example

The following example shows the results of a query on the model.config table from the service Model being assigned to a previously declared variable of datatype RecordList called configData.

```
configData = RetrieveOQLFromService(
            "select * from model.config;",
            "Model"
            );
```

Example

The following example shows the results of a query on the services.unManaged table from the service Ctrl.

```
RecordList results = RetrieveOQLFromService( "select * from services.unManaged;",
"Ctrl" )

for (results)
{
    Record current = GetInScopeRecord();
}
```

RetrieveSingleOQL()
`RetrieveSingleOQL();` rule is a version of the `RetrieveOQL();` rule that returns only the first result of the OQL query, regardless of how many records are returned.An optional record can be passed in to be added to the scope of the OQL query.

Syntax

The `RetrieveSingleOQL();` statement uses following syntax.

```
RetrieveSingleOQL( oql string [, optional record] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 34. Arguments of RetrieveSingleOQL()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`oql string`	OQL command to execute. This can reference members of the optional record, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	Yes	Yes
`optional record`	If present, this is added so that it can be dereferenced in the OQL string.	No	Yes	No

Example

The following example shows how the `RetrieveSingleOQL()` rule is used to return only the first result of the OQL query.

```
Record singleRow = RetrieveSingleOQL( "select * from disco.config;" )
```

RollbackSQLTransaction()
The `RollbackSQLTransaction();` rule rolls back a transaction in a specified database.

Syntax

The `RollbackSQLTransaction();` statement uses following syntax.

```
RollbackSQLTransaction( database identifier );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 35. Arguments of RollbackSQLTransaction()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`database identifier`	Specifies the database on which the rollback transaction is to be performed; for example, DNCIM, NCIM, NCMONITOR.	No	No	No

Example

The following example shows how the rule is used.

```
StartSQLTransaction( "DNCIM" );
        text entityName = NULL;
        foreach(entityNames)
        {
            entityName = eval(text,'&m_Name');
            ExecuteStitcher('PopulateDNCIMObject', entityName, domainId,
dynamicDiscoNode);
        }
        delete(entityNames);
RollbackSQLTransaction( "DNCIM" );
```

Related reference

CommitSQLTransaction()
The `CommitSQLTransaction();` rule commits a transaction in a specified database.

StartSQLTransaction()
The `CommitSQLTransaction();` rule starts a commit transaction in a specified database.

SendOQLtoService()
The `SendOQLToService();` rule executes an OQL query, which is not expected to return data, against another process.

Syntax

The `SendOQLToService();` statement uses following syntax.

```
SendOQLToService( service name, oql string );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 36. Arguments of SendOQLToService()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`service name`	Name of the process to send results to. Available strings can be found by running the command `ncp_oql -options`.	Yes	No	No
`oql string`	OQL command to execute. This can reference members of the optional record, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	No	Yes

Example

The following example shows how the rule is used to perform an insert into the ncp_ctrl `services.unmanaged` table.

```
SendOQLToService(
        "Ctrl",
        "insert into services.unManaged( serviceName, servicePath, argList)
        values
        ("ping", "/usr/sbin/", ['1,2,3,4']);");
        );
```

Example

The following example shows how the rule is used to perform an update operation.

```
SendOQLToService( "Model",
    "update model.config set DiscoveryUpdateMode = eval(int '$isRediscovery');" );
```

SendAllOQLToService()
The `SendAllOQLToService();` stitcher rule sends all the records in a variable of type `RecordList` to another service. Use this rule to more efficiently send a batch of OQL instructions over the network rather than several individual OQL requests.

Example

The `SendAllOQLToService();` statement uses following syntax.

```
SendAllOQLToService(service_name,variable,oql string,test stitcher );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 37. Arguments of SendAllOQLToService()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`service name`	Name of the process to send results to. Available strings can be found by running the command `ncp_oql -options`.	Yes	No	No
`variable`	Variable to send the records from.	No	Yes	No
`oql string`	OQL command to execute. This can reference members of the optional record, if present. **Note:** OQL queries must not be terminated with a semi-colon.	Yes	No	Yes
`test stitcher`	Stitcher to invoke for each record inserted.	Yes	No	No

Example

In the following example, `SomeScript.pl` is a script that you want to run against multiple entities. `ScriptDataFilter` is a stitcher that verifies that the arguments are correct.

```
SendAllOQLToService(
   "Ctrl",                      // Send OQL to this service
   scriptTargets,                   // Send the records from this variable

   "insert into services.unManaged
       ( serviceName, servicePath, argList, logFile )
   values
       ( 'SomeScript.pl', '$NCHOME/precision/scripts/perl/scripts',
eval(list type text, '$scriptTargetArgList') , 'someScript.log' );",

   "ScriptDataFilter"        // Invoke this stitcher for each record
        );
```

SetReturnValue()
The `SetReturnValue();` sets a return value for a stitcher.

Syntax

The `SetReturnValue();` statement uses the following syntax.

```
SetReturnValue( return value );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 38. Arguments of SetReturnValue()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`return value`	The value to return from the stitcher.	Yes	Yes	Yes

Example

The following example shows how the `SetReturnValue()` rule is used to return a value.

```
text entityName = "router 1";
SetReturnValue( entityName );
```

StandardiseIPv6()
The `StandardiseIPv6();` stitcher rule converts a textual (colon-notation) IPv6 address into standard notation. This includes, for example, stripping the longest string of zeroes down to '::'.

Syntax

The `StandardiseIPv6();` statement uses the following syntax.

```
StandardiseIPv6 ( ipv6 string );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 39. Arguments of StandardiseIPv6()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`ipv6 string`	Textual (colon-notation) IPv6 address	Yes	Yes	No

Example

The following example shows how the `StandardiseIPv6()` rule is used to convert a textual (colon-notation) IPv6 address into standard notation.

```
text ipAddress = eval(text, '&m_UniqueIPv6Address');
ipAddress = StandardiseIPv6( ipAddress );
```

StartSQLTransaction()
The `CommitSQLTransaction();` rule starts a commit transaction in a specified database.

Syntax

The `StartSQLTransaction();` statement uses following syntax.

```
StartSQLTransaction( database identifier );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 40. Arguments of StartSQLTransaction()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`database identifier`	Specifies the database on which the commit transaction is to be performed; for example, DNCIM, NCIM, NCMONITOR.	No	No	No

Example

The following example shows how the rule is used.

```
StartSQLTransaction( "DNCIM" );
        text entityName = NULL;
        foreach(entityNames)
        {
            entityName = eval(text,'&m_Name');
            ExecuteStitcher('PopulateDNCIMObject', entityName, domainId,
dynamicDiscoNode);
        }
        delete(entityNames);
CommitSQLTransaction( "DNCIM" );
```

Related reference

CommitSQLTransaction()
The `CommitSQLTransaction();` rule commits a transaction in a specified database.

RollbackSQLTransaction()
The `RollbackSQLTransaction();` rule rolls back a transaction in a specified database.

StitcherTimeCheck()
The `StitcherTimeCheck();` stitcher rule prints a message to stdout. From version 3.8 onwards, this message appears in the .trace log file.

Syntax

The `StitcherTimeCheck();` statement uses the following syntax.

```
StitcherTimeCheck ( finished item, started item, [optional percentage complete] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 41. Arguments of StitcherTimeCheck()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`finished item`	A string describing the item of processing that has been completed.	Yes	No	No
`started item`	A string describing the item of processing that has been started.	Yes	No	No
`optional percentage complete`	An integer indicating the what percentage of processing of has completed.	Yes	No	No

Example

The following example shows how the `StitcherTimeCheck()` rule is used to indicate that the working entities database table has been populated, containment is being built, and the overall process is 20% complete.

```
# .
StitcherTimeCheck( 'Working Entities Table Population', 'Containment Building', 20 )
```

Discovery stitcher rules

Use these stitcher rules for reference when you are working with discovery stitchers.

AnalyzeSQLStats()

The `AnalyzeSQLStats();` stitcher rule analyzes an SQLite database or a single specified table in that database and ensures that SQLite has up-to-date statistics on its database tables from which it can make efficient SQL query plans. This ensures that any subsequent SQL select queries made using the data in the tables are efficient.

In version 4.2 the only database that uses the SQLite database platform is the embedded relational database, known as Discovery NCIM (DNCIM). The `AnalyzeSQLStats();` rule is called from within the DNCIM stitchers, once most of the DNCIM data has been inserted, but before discovery postprocessing takes place. Note that if the database platform is something other than SQLite then this stitcher rule does nothing.

Syntax

The `AnalyzeSQLStats();` statement uses the following syntax.

```
AnalyzeSQLStats ( DNCIM , database name or table name );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 42. Arguments of AnalyzeSQLStats();				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`database name or table name`	Name of database or database table to analyze.	Yes	No	No

Example

The following examples shows how the `AnalyzeSQLStats();` rule is used in different discovery stitchers:

Table 43. Examples of use of the AnalyzeSQLStats(); stitcher rule

Stitcher	Sample code	Description
PopulateDNCIM.stch	`AnalyzeSQLStats('DNCIM','dncim');`	Performs analysis of the entire DNCIM database.
PopulateDNCIM_BGPTopology.stch	`AnalyzeSQLStats('DNCIM', 'bgpService');`	Performs analysis of the DNCIM table bgpService only.
	`AnalyzeSQLStats('DNCIM', 'bgpEndPoint');`	Performs analysis of the DNCIM table bgpEndPoint only.

DiscoReadConfig()
The `DiscoReadConfig` rule tells the Discovery Engine, ncp_disco, to reread its configuration files.

Syntax

By default this rule is used in the FullDiscovery stitcher, although it can be defined in any stitcher. This rule instructs the Discovery Engine, ncp_disco, to reread its configuration files whenever you launch a full discovery. The rule determines which configuration files to update by reading the records in the Discovery engine database table disco.dynamicConfigFiles. For more information on the disco.dynamicConfigFiles database table, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

The following syntax shows how to use the `DiscoReadConfig` rule.

```
DiscoReadConfig();
```

Related reference

disco.dynamicConfigFiles table
The dynamicConfigFiles table stores the names of configuration files that must be reread each time a full discovery is launched.

DncimRecordsDone()
The `DncimRecordsDone();` rule notifies the discovery that a RecordToDNCIMDb() rule session has completed and instructs the discovery to commit any outstanding SQL actions.

Syntax

The `DncimRecordsDone();` statement uses following syntax.

```
DncimRecordsDone(  );
```

Arguments

This stitcher rule takes no arguments.

Example

The following example shows how the rule is used to perform an insert into the ncp_ctrl `services.unmanaged` table.

```
// Add custom data from ModelNcimDb.cfg
            RecordToDncimDb();

// Finish the record to dncim transaction
            DncimRecordsDone();
```

Related reference

RecordToDncimDb()

The `RecordToDncimDb();` takes a record and passes it through the mappings in the ModelNcimDb.cfg and DbEntityDetails.cfg configuration files. These mappings are used to populate multiple dNCIM tables and create new objects.

DiscoRefresh()

The `DiscoRefresh();` stitcher rule sends a refresh message to the Helper server, finders, or agents connected to the discovery process. The effect of the refresh message will vary depending on the process and the arguments.

Syntax

The `DiscoRefresh();` statement uses the following syntax.

```
DiscoRefresh ( application name , optional arguments );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 44. Arguments of DiscoRefresh();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`application name`	The name of the application to refresh. This can be one of the following: • Helper • PingFinder • FileFinder • DBEntryFinder • CollectorFinder • Name of any discovery agent	Yes	No	No
`optional arguments`	Application-specific arguments to refresh. For example, for the Ping finder, it is possible to refresh the scope. For the Helper server, it is possible to refresh a specific helper, such as the SnmpHelper. For more information, see the Example section.	Yes	Yes	Yes

Applications

This section provides examples showing how to refresh the following applications:

- Helper server
- Finders
- Agents

Helper server

The Helper server can store data from helper requests. Consequently, if multiple agents request the same data, the data is retrieved from the device once only. By refreshing the Helper server, you are deleting existing stored Helper server data. You can refresh Helper server data at the following levels:

- Delete all stored data.
- Delete all stored data for a specific helper; for example, the SNMP helper.
- Delete all stored data for a specific device with a specific helper.

Examples of each of these refresh levels are provided below.

Example: refreshing the Helper server

The following example shows how the `DiscoRefresh();` rule can be used to refresh the Helper server. This causes the helper server to delete any stored entity data.

```
#              Refresh all the helper server stored helper data.
              DiscoRefresh('Helper');
```

Example: refreshing stored helper server data for specific helpers

The following example shows how the `DiscoRefresh();` rule can be used to refresh stored helper server data for specific helpers, by providing the name of the helper to be refreshed.

```
DiscoRefresh('Helper','SnmpHelper');
DiscoRefresh('Helper','TelnetHelper');
DiscoRefresh('Helper','DNSHelper');
DiscoRefresh('Helper','ARPHelper');
DiscoRefresh('Helper','PingHelper');
DiscoRefresh('Helper','XmlRpcHelper');
```

Example: removing stored Helper server data for a specific device with a specific helper

The following example shows how the `DiscoRefresh();` rule can be used to specify a specific filter to use to remove stored Helper server data for a specific device with a specific helper. This action is usually based on the `m_HostIp` field. For more information, see `$NCHOME/etc/precision/DiscoHelperServerSchema.cfg` configuration file.

```
DiscoRefresh('Helper','SnmpHelper','m_HostIp', '1.2.3.4' );
```

Finders

The following examples show how the `DiscoRefresh();` rule can be used with finders.

Example: refreshing the finders

The following example shows how the `DiscoRefresh();` rule can be used to refresh the finders. If only the finder name is provided then this results in the finder rereading its configuration and sending any resulting finder inserts. This is commonly used to tell the Ping or File finder to reread its seed configuration and process the IP addresses found.

```
DiscoRefresh('PingFinder');
DiscoRefresh('FileFinder');
DiscoRefresh('CollectorFinder');
```

Example: rereading the scope setting

The following example shows how the `DiscoRefresh();` rule can be used to reread the scope setting.

```
DiscoRefresh('PingFinder','scope');
```

Example: requesting the refresh of a particular subnet

The following example shows how the `DiscoRefresh();` rule can be used to request the refresh of a particular subnet.

```
DiscoRefresh('PingFinder','1.2.3.0','255.255.255.0');
```

Example: refreshing the Collector finder

The following example shows how the `DiscoRefresh();` rule can be used to refresh the Collector finder. In this example, a request is sent to the Collector finder to refresh a given EMS host.

```
DiscoRefresh('CollectorFinder', '1.2.3.4');
```

Example: doing a partial discovery refresh every ten minutes

The following example shows how a timed stitcher can be used together with the `DiscoRefresh();` rule acting on the Database finder to do a partial discovery refresh every ten minutes.

Note: For the Database finder, the `optional arguments` that can be included within the `DiscoRefresh();` rule are as follows. In each of these cases, the queries and trigger types are defined in the DiscoDBEntryFinderQueries.cfg configuration file.

- If no `optional arguments` value is specified, then run queries associated with trigger type 1.
- PARTIAL: run queries associated with trigger type 2.
- FORCED: run queries associated with trigger type 3.

```
UserDefinedStitcher
{
    StitcherTrigger
    {
        // Activate every minute
        ActOnTimedTrigger(( m_IntervalSeconds ) values ( 60 ) ; );
    }

    StitcherRules
    {
        RecordList discoStatus = RetrieveOQL(
            "select * from disco.status;"
        );

        int phase = -1;

        foreach(discoStatus)
        {
            phase = eval(int, '&m_Phase');
        }
        delete(discoStatus);

        if(phase >= 0  AND  phase <= 1)
        {
            DiscoRefresh
            (
                "DBEntryFinder", "PARTIAL"
            );
        }
    }
}
```

Agents

The following example shows how the `DiscoRefresh();` rule can be used with finders.

Example: having an agent reread discovery scope

The agents have very little state as they process each entity on a case by case basis. As a result the agents rarely need to be refreshed, with the exception of agents which require knowledge of discovery scope in order to function. An example of such an agent is the IpRoutingTable agent, which uses the

discovery scope to determine which sections of a potentially huge routing table to attempt to download. The following example shows how the `DiscoRefresh();` rule can be used to refresh the IpRoutingTable agent, by having the agent reread the discovery scope.

```
DiscoRefresh( 'IpRoutingTable', 'scope');
```

DiscoRetrieveClass()
The `DiscoRetrieveClass();` stitcher rule is used to retrieve the class of a discovered device during the discovery process, so that users can determine device type while the discovery is still running.

This rule is used at the following points in the discovery process:

- During the data collection stage the rule is used by the AssocAddressRetProcessing.stch stitcher. This stitcher uses the output from this rule to update the AssocAddress returns table with class information. This provides the information needed to display device type while discovery is still running.
- During the data processing stage the rule is used by the AddClass stitcher to set the class of the chassis objects.

Syntax

The `DiscoRetrieveClass();` statement uses the following syntax.

```
DiscoRetrieveClass ( record position , source format );
```

The information in the output class record is class ID, class name, and the name of the visual icon used to represent the class.

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 45. Arguments of DiscoRefresh()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`record position`	Position in the record stack from which to extract the record. This is equivalent to what you would retrieve using an `eval` statement with a single ampersand.	Yes	No	No
`source format`	The source format of the class data. Possible values include workingEntities and ncim. The DiscoRetrieveClass(); rule translates the data from the source format into the format expected by the AOC files.	Yes	No	No

Example: Helper server

The following example shows how the `DiscoRetrieveClass();` rule can be used to retrieve class data and load it into a variable.

```
Record  classRec = NULL;
classRec = DiscoRetrieveClass(0, "workingEntities");
```

DiscoSendOQLToFinder()
The `DiscoSendOQLToFinder();` stitcher rule sends OQL results to the finder subprocess from the discovery. This is commnly used as part of the feedback process, where the ping finder is seeded with subnets and IP addresses discovered during the course of the network discovery.

Syntax

The `DiscoSendOQLToFinder();` statement uses the following syntax.

```
DiscoSendOQLToFinder ( finder name , oql string );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 46. Arguments of DiscoSendOQLToFinder();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`finder name`	The name of the finder to send the OQL results to. This can take one of the following values: • PingFinder • FileFinder • CollectorFinder	Yes	No	No
`oql string`	The OQL string to send to the finder.	Yes	Yes	Yes

Example

The following example shows how the `DiscoSendOQLToFinder();` rule is used to seed the ping finder with a newly discovered subnet on which to perform a ping sweep.

```
# Seed the ping finder with a newly discovered subnet to ping sweep
DiscoSendOQLToFinder
(
              "PingFinder",
              "insert into pingFinder.pingRules
              (
                            m_Address,
                            m_RequestType,
                            m_NetMask,
                            m_Protocol
              )
              values
              (
                            eval(text, '&m_LocalNbr->m_Subnet'),
                            eval(int, '$PingSubnet'),
                            eval(text, '&m_LocalNbr->m_SubnetMask'),
                            eval(int, '$protocol')
              );"
);
```

DncimUpdate()
The `DncimUpdate();` rule updates a record in a specified table in the dNCIM database and sets a flag so that the updated row will be broadcast in the next call to the `BroadcastToModel();` rule. The `DncimUpdate();` rule can also be called omitting the optional record to force a broadcast of the row even if no other changes are made. This is useful when a row has been removed as part of a cascaded delete.

Syntax

The `DncimUpdate();` statement uses the following syntax.

```
DncimUpdate ( entityId, tableName [, optional record ] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 47. Arguments of DncimUpdate();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId`	Integer identifying the entity related to the update operation.	Yes	No	No
`tableName`	Name of the table in the dNCIM database schema to update with the contents of the record.			
`optional record`	Updates to apply to the row in the `tableName` table, if supplied.	Yes	No	No

Example

The following example updates the displayLabel field of the entityData table.

```
Record serviceLabelUpdate;
      @serviceLabelUpdate.displayLabel = serviceLabel;

      DncimUpdate( ospfServiceEntityId, "entityData", serviceLabelUpdate );
```

Example

The following example performs no update, but forces the collection to be broadcast in the next call to the `BroadcastToModel();` rule.

```
      DncimUpdate( entityId, "collects" );
```

Returns

This rule does not return any values.

EnumerationLookup()

The `EnumerationLookup();` rule retrieves a value from the DNCIM enumeration table. The data in the enumeration table is downloaded when the Discovery engine, ncp_disco, is first started and then accessed directly by the rule, thereby enabling fast data retrieval.

The enumerations available for lookup are determined by the entry in the dbModel.access table within the ModelNcimDb.cfg configuration file; for example:

```
insert into dbModel.access
(
    EnumGroupFilter,
    TransactionLength,
    WebTopDataSource
)
values
(
    "enumGroup in ('ASN' , 'sysServices', 'ifAdminStatus', 'ifOperStatus',
'sysServices', 'ifType', 'ifOperStatusToOperationalStatus',
'entPhysicalClass', 'cefcFRUPowerAdminStatus', 'cefcFRUPowerOperStatus',
'TruthValue','TruthValueString', 'entSensorType', 'entSensorScale',
'entSensorStatus', 'cefcModuleAdminStatus', 'cefcModuleOperStatus', 'ipForwarding',
```

```
'cefcPowerRedundancyMode', 'EntityType', 'ospfIfState', 'ospfIfType',
'dot3StatsDuplexStatus', 'accessProtocol', 'cdmDuplex', 'OperationalStatusEnum')",
    0,
    "NCOMS"
);
```

Syntax

The `EnumerationLookup();` statement uses the following syntax.

```
EnumerationLookup ( evalClause ] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 48. Arguments of EnumerationLookup();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`evalClause`	Defines the details for the lookup eval statement.	Yes	For the first argument.	No

The full definition of `evalClause` is as follows:

```
resultStr = EnumerationLookup( eval(text, 'LOOKUP( $key, &&enum _group
[ , optional_default ] )') );
```

Where:

- *key* is the value to use to perform the enumeration string lookup operation.
- *enum _group* is the name of enumeration group to look in.
- *optional_default* is the value to use if the lookup operation returns null.

Example

The following examples show how to use this rule.

```
text protocol = NULL;
protocol = EnumerationLookup( eval(text, 'LOOKUP( &m_LocalNbr->m_Protocol,
&&accessProtocol, `IPv4`)') );

int keyVal = 1; text truthStr = NULL; truthStr =
EnumerationLookup( eval(text, 'LOOKUP( $keyVal, &&TruthValueString )') );

int ifType = 6; text ifTypeString = NULL; ifTypeString =
EnumerationLookup( eval(text, 'LOOKUP( $ifType, &&ifType )') );
```

Note: This rule does not use the standard record stack. Normally the ampersand sign & accesses the record at the top of the stack, double ampersand && the second from top, triple ampersand &&& the third, and so on. With this rule the record stack used only contains two records; therefore, the ampersand & refers to the record on the top of the normal stack as before, but double ampersand && refers to the record containing the static enumeration data downloaded when discovery was first started.

Returns

This rule returns the text string relating to the desired enumeration.

RecordToDncimDb()
The `RecordToDncimDb();` takes a record and passes it through the mappings in the ModelNcimDb.cfg and DbEntityDetails.cfg configuration files. These mappings are used to populate multiple dNCIM tables and create new objects.

Syntax

The `RecordToDncimDb();` statement uses the following syntax.

```
RecordToDncimDb ( [optional record] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 49. Arguments of RecordToDncimDb();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`optional record`	A record of custom data retrieved from a custom agent or collector. If present, this record is passed to custom DNCIM tables using the mappings defined in the ModelNcimDb.cfg and DbEntityDetails.cfg configuration files. If no record is passed in as an argument then the record on the top of the stack (i.e,. the record currently in scope), is used.	No	Yes	No

Related reference

DncimRecordsDone()
The `DncimRecordsDone();` rule notifies the discovery that a RecordToDNCIMDb() rule session has completed and instructs the discovery to commit any outstanding SQL actions.

StitcherProfiling()
The `StitcherProfiling();` stitcher rule sets the discovery process to log the time each stitcher takes to complete and the number of executions for each stitcher during a discovery cycle. The logging results help diagnose discovery issues. This rule is set in the FinalPhase stitcher.

Syntax

The `StitcherProfiling();` statement uses the following syntax:

```
StitcherProfiling ( profile action, [optional additional variable] );
```

The information in the output provides the name of the stitcher, the number of times it ran, and the total run time in seconds.

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 50. Arguments of StitcherProfiling();

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`profile action`	Can take the following values: • LOG: Triggers the logging of the time each stitcher takes to complete. • RESET: Resets the data logged using the LOG option to zero. The data is not accumulated and the collection starts again at the next discovery. • SIZE: Logs the current size of the process.	Yes	No	No
`[optional additional variable]`	Dependent on the profile action: • For LOG: A stitcher name can be supplied to log profiling data for a single stitcher. • For RESET: A stitcher name can be supplied to reset the log for a single stitcher. • For SIZE: An optional tag to include in the log message.	Yes	No	No

The following examples show how to set the `StitcherProfiling();` rule in different ways.

To use the rule to trigger the logging of profiling data on all stitchers:

```
StitcherProfiling('LOG');
```

To use the rule to reset the profiling data on all stitchers to zero, ready to log data only for the next discovery:

```
StitcherProfiling('RESET');
```

To use the rule to log profiling data only for a specific stitcher (using an additional optional argument):

```
StitcherProfiling('LOG', 'RecreateAndSendTopology');
```

To use the rule to log the current size of the process:

```
StitcherProfiling('SIZE');
```

To use the rule to log the current size of the process, and include an additional tag to add to the end of the log message:

```
StitcherProfiling('SIZE', 'before building the layers');
```

Event Gateway stitcher rules

Use these stitcher rules for reference when you are working with Event Gateway stitchers. These rules are only available within the Event Gateway, ncp_g_event and its plugins.

GwCollects()

The `GwCollects();` rule retrieves a list of entities directly collected by a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.collects`.

Syntax

The `GwCollects();` statement uses following syntax.

```
GwCollects ( entityId or entityName );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 51. Arguments of GwCollects()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return entities collected by this entity.	Yes	Yes	Yes

Example

The following example finds all entities collected by the specified entity and then prints out the results.

```
int entityId = eval( int, '&NmosEntityId' );

# Find all entities collected by the given entityId...
RecordList collectedEntities = GwCollects( entityId );

# ...and iterate through the returned results
Record collectsRow;
foreach ( collectedEntities )
{
         collectsRow = GetInScopeRecord();
         PrintRecord( collectsRow  );
}
```

Returns

This rule returns a list of records. Each record contains information about a single collected entity, as contained in the `ncimCache.collects` table.

The following snippet shows an example of the results returned by this rule.

```
{
        ENTITYNAME='IGMP_ENDPOINT_istanbul-asbr-cr26.tk.eu.test.lab[ Fa0/0 ]';
        SEQUENCE=NULL;
},
{
        ENTITYNAME='IGMP_ENDPOINT_pe6-cr38.core.eu.test.lab[ Vl1 ]';
        SEQUENCE=NULL;
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

GwConnects()
The `GwConnects();` rule retrieves a list of entities directly connected to a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.connects`. A limitation on this rule is that it does not retrieve connections to contained entities. For example, if the entity passed to the rule represents a chassis, connections to interfaces within that chassis will not be returned.

Syntax

The `GwConnects();` statement uses following syntax.

```
GwConnects ( entityId or entityName, [optional topology name] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 52. Arguments of GwConnects()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return entities connected to this entity.	Yes	Yes	Yes
`optional topology name`	Name of a topology as specified in the entityData table.	Yes	No	No

Example

The following example finds all connections to a specified entity and then prints out all of the results.

```
text entityName = eval( text, '&EntityName' );

# Find all connections to the given entityName...
RecordList connectedEntities = GwConnects( entityName );

# ...and iterate through the returned results
Record singleConnection;
foreach ( connectedEntities )
{
        singleConnection = GetInScopeRecord();
        PrintRecord( singleConnection );
}
```

Example

The following example finds all connections to a specified entity within a specific topology.

```
# Find all connections to the given entityName within a specific topology
RecordList specificConnectedEntities = NULL;
specificConnectedEntities = GwConnects( entityName, 'RouterLinksTopology' );
```

Returns

This rule returns a list of records. Each record contains information about a single connected entity, as contained in the `ncimCache.connects` table.

The following snippet shows an example of the results returned by this rule.

```
{
        TOPOENTITYNAME='RouterLinksTopology';
        UNIDIRECTIONAL=0;
        ENTITYNAME='freddy[ 0 [ 1 ] ]';
},
{
        TOPOENTITYNAME='RelatedToTopology';
        UNIDIRECTIONAL=0;
        ENTITYNAME='bob[ 0 [ 1 ] ]';
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

GwContains()
The `GwContains();` rule retrieves a list of entities directly contained by a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.contains`. No recursion is performed. For example, if a chassis contains some cards, and those cards contain interfaces, the result of running this rule against the chassis will be a list of cards.

Syntax

The `GwContains();` statement uses following syntax.

```
GwContains ( entityId or entityName );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 53. Arguments of GwContains()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return entities contained within this entity.	Yes	Yes	Yes

Example

The following example retrieves a list of entities directly contained by a specified entity and then prints out all of the results.

```
text entityName = eval( int, '&EntityName' );

# Find all entities contained by the given entityName...
RecordList containedEntities = GwContains( entityName );
```

```
# ...and iterate through the returned results
Record containsRow;
foreach ( containedEntities )
{
         containsRow = GetInScopeRecord();
         PrintRecord( containsRow);
}
```

Returns

This rule returns a list of records. Each record contains information about a single contained entity, as contained in the `ncimCache.contains` table.

The following snippet shows an example of the results returned by this rule.

```
{
        ENTITYNAME='ny-p2-cr28.na.test.lab[ Fa1/7 ]';
        UPWARDCONNECTION=1;
},
{
        ENTITYNAME='ny-p2-cr28.na.test.lab[ Fa1/8 ]';
        UPWARDCONNECTION=1;
},
{
        ENTITYNAME='ny-p2-cr28.na.test.lab[ Fa1/12 ]';
        UPWARDCONNECTION=1;
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

GwDependency()

The `GwDependency();` rule retrieves a list of dependencies upon a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.dependency`.

Syntax

The `GwDependency();` statement uses following syntax.

```
GwDependency ( entityId or entityName, [optional dependency identifier] );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 54. Arguments of GwDependency()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return entities dependent upon this entity.	Yes	Yes	Yes
`optional dependency identifier`	Instructs the rule to only retrieve this type of dependency.	Yes	No	No

Example

The following example finds all dependent entities and then prints out all of the results.

```
int entityId = eval( text, '&NmosEntityId' );
# Find all depencies on the given entityName...
RecordList dependentEntities = GwDependency( entityId );

# ...and iterate through the returned results
Record singleDependent;
foreach ( dependentEntities )
{
         singleDependent = GetInScopeRecord();
         PrintRecord( singleDependent );
}
```

Example

The following example finds all dependent entities of a specific type.

```
# Find all dependents of a specific type
RecordList specificDependents = GwDependency( entityId , 1 );
```

Returns

This rule returns a list of records. Each record contains information about a single dependent entity, as contained in the `ncimCache.dependency` table.

The following snippet shows an example of the results returned by this rule.

```
{
        DEPENDENCYTYPE=1;
        ENTITYNAME='IPMRoute_UpstreamRoute_core2-cs35.core.eu6.test.lab_
        (0.0.0.0,239.255.255.255)';
},
{
        DEPENDENCYTYPE=1;
        ENTITYNAME='IPMRoute_UpstreamRoute_istanbul-asbr-cr26.tk.eu.test.lab
        [ Fa0/1 ]_(0.0.0.0,239.255.255.255)';
},
{
        DEPENDENCYTYPE=1;
        ENTITYNAME='IPMRoute_DownstreamRoute_istanbul-asbr-cr26.tk.eu.test.lab
        [ Fa0/0 ]_(0.0.0.0,239.255.255.255)_239.255.255.255';
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database

This database stores topology updates from DNCIM.

GwEnrichEvent()

The `GwEnrichEvent();` updates fields in an event. Any data can be added to an event using this rule; however, the only fields which are permitted to update the ObjectServer alerts.status table are those fields that are listed in the outgoing field filter, as defined in the in the FieldFilter section of the nco2ncp table within the EventGatewaySchema.cfg configuration file.

Syntax

The `GwEnrichEvent();` statement uses the following syntax.

```
GwEnrichEvent ( [optional serial number], enrichedFields );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 55. Arguments of GwEnrichEvent()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`optional serial number`	Identifies the event to be updated. If no serial number is supplied, it is assumed that the current top-level in-scope event is being updated. If iterating through a loop (for example, the contents of a RecordList), the event is still the top-level event that was in-scope at the start of the stitcher, and not the current in-scope record within the loop. **Note:** The Event Gateway stitchers always have the event as the top-level in-scope record.	Yes	Yes	Yes
`enrichedFields`	Record listing the fields to use to enrich the event. These fields are added to the event or update existing fields in the event.	Yes	Yes	Yes

Returns

This stitcher does not return any values.

Updating the current in-scope event

The following example shows how the `GwEnrichEvent();` rule is used to update the current in-scope record.

Both fields specified as part of the enrichment record are added to the current in-scope record. However, only the MyCustomField column of the alerts.status table is actually updated in the ObjectServer because the FieldUsedByRca field is not listed as a permitted field in the outgoing field filter.

```
// define the fields to add or update
Record enrichment;
@enrichment.MyCustomfield = "any value I choose for this text field";
@enrichment.FieldUsedByRca = 59;

GwEnrichEvent( enrichment );
```

Updating an event with a specified serial number

The following example shows how the `GwEnrichEvent();` rule is used to update an event with a specified serial number. This example works exactly as the previous example, except that the top-level in-scope record is modified if and only if it has a serial number of 345. If the event does not have serial number 345, then the alerts.status table is updated for that seial number, but the in-scope record remains unaltered.

```
// define the fields to add or update
Record enrichment;
@enrichment.MyCustomfield = "any value I choose for this text field";
@enrichment.FieldUsedByRca = 59;
```

```
GwEnrichEvent( 345, enrichment );
```

GwEntityData()
The `GwEntityData();` rule provides a simplified way of looking up topology data in NCIM cache. This rule performs topology lookups in the ncimCache.entityData table.

Syntax

The `GwEntityData();` statement uses the following syntax.

```
GwEntityData ( entity ID or entity name );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 56. Arguments of GwEntityData()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entity ID`	The integer entity ID is matched against the ENTITYID field in the ncimCache.entityData table.	Yes	Yes	Yes
`entity name`	The text entity name is matched against the ENTITYNAME field in the ncimCache.entityData table.	Yes	Yes	Yes

For more information on the ncimCache tables, see section *NCIM cache* in the *IBM Tivoli Network Manager Reference*.

Returns

The `GwEntityData();` statement returns a row from the ncimCache.entityData table if a result was found.

Performing a lookup based on the value of in-scope data

The following example shows how the `GwEntityData();` rule is used to perform a topology lookup in NCIM cache based on the value of in-scope data.

In this example the nmosEntityId variable is loaded with the value of a field from an in-scope event. The `GwEntityData()` rule then uses the value of the nmosEntityId to perform a topology lookup and load the results of the lookup into the entity record.

```
int nmosEntityId = eval(int, '&NmosEntityId');
Record entity = GwEntityData( nmosEntityId );
```

Performing a lookup based on a supplied string

The following example shows how the `GwEntityData();` rule is used to perform a topology lookup in NCIM cache based on the value of a supplied string.

In this example the `GwEntityData()` rule uses the value of a supplied string to perform a topology lookup and load the results of the lookup into the entity record.

```
entity = GwEntityData( "device_name[ 0 [ 1 ] ]" );
```

GwHostedService()
The `GwHostedService();` rule retrieves a list of entities directly connected to a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.hostedService`.

Syntax

The `GwHostedService();` statement uses following syntax.

```
GwHostedService ( entityId or entityName );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 57. Arguments of GwHostedService()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return services hosted by this entity.	Yes	Yes	Yes

Example

The following example services hosted by a specified entity and then prints out the results.

```
text entityName = eval( int, '&EntityName' );

# Find all services hosted by the given entityName...
RecordList hostedServices = GwHostedService( entityName );

# ...and iterate through the returned results
Record hostedServiceRow;
foreach ( hostedServices )
{
         hostedServiceRow = GetInScopeRecord();
         PrintRecord( hostedServiceRow );
}
```

Returns

This rule returns a list of records. Each record contains information about a single hosted service, as contained in the `ncimCache.hostedService` table.

The following snippet shows an example of the results returned by this rule.

```
{
        ENTITYNAME='PIM_SERVICE_salida-abr-cr36.na.test.lab';
},
{
        ENTITYNAME='IPMRoute_Service_salida-abr-cr36.na.test.lab';
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database

This database stores topology updates from DNCIM.

GwIpLookup()
The `GwIpLookup();` rule allows rapid and efficient topology lookups for the entity that implements a given IP address or DNS name. This entity is often an interface, but, if no SNMP access is available to a device, it could be the related chassis.

Syntax

The `GwIpLookup();` statement uses the following syntax.

```
GwIpLookup ( IP address or DNS name );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 58. Arguments of GwIpLookup()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`IP address`	Textual IP address. This value is matched against the relevant field in the ncimCache.entityData table.	Yes	Yes	Yes
`DNS name`	Textual DNS name. This value is matched against the relevant field in the ncimCache.entityData table.	Yes	Yes	Yes

For more information on the ncimCache tables, see section *NCIM cache* in the *IBM Tivoli Network Manager Reference*.

Returns

The `GwIpLookup();` statement returns a row from the ncimCache.entityData table if a result was found.

Performing a lookup based on a DNS name

The following example shows how the `GwIpLookup();` rule is used to perform a topology lookup in NCIM cache based on the value of a supplied DNS name.

In this example the `GwIpLookup()` rule uses the value of a supplied DNS name to perform a topology lookup and load the results of the lookup into the entity record.

```
text dnsName = "anydevice.com";
implementingEntity = GwIpLookup( dnsName );
```

Performing a lookup based on an IP address

The following example shows how the `GwIpLookup();` rule is used to perform a topology lookup in NCIM cache based on the value of a supplied IP address.

In this example the `GwIpLookup()` rule uses the value of a supplied IP address to perform a topology lookup and load the results of the lookup into the entity record.

```
Record implementingEntity = GwIpLookup( "9.196.131.49" );
```

GwIpLookupUsing()
The `GwIpLookupUsing();` rule performs the same operation as the `GwIpLookup()` rule, except that a field within the current top-level in-scope event is used to perform the lookup.

Syntax

The `GwIpLookupUsing();` statement uses following syntax.

```
GwConnects ( event field name);
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 59. Arguments of GwIpLookupUsing()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`event field name`	Instructs the rule to extract the value of this field from the current event, and use it to look for the entity in the topology.	Yes	Yes	Yes

Example

The following example looks up an entity based on a supplied event field name.

```
Record implementingEntity= GwIpLookupUsing( "LocalNodeAlias" );
```

GwMainNodeLookup()
The `GwMainNodeLookup();` rule performs topology lookups for main nodes, based on a limited number of fields.

This rule performs topology lookups for main nodes, based on the following fields:

- IP address (as listed in the ipEndPoint table)
- entityName (as given in the entityData table)
- DNS name (as given in the ipEndPoint table)
- sysName (as given in the chassis table)
- entityId (as given in the entityData table)

The rule returns the row from the ncimCache.entityData table if a result was found.

Syntax

The `GwMainNodeLookup();` statement uses following syntax.

```
GwMainNodeLookup ( identifier);
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 60. Arguments of GwMainNodeLookup()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`identifier`	Text or integer identifier used to look up the main node. • If an integer value is specified, then the lookup operation assumes this is an entityId. • If a string field is specified, then the lookup operation assumes this is an IP address, DNS name, sysName or entityName. In each case, an appropriate lookup operation will be performed.	Yes	Yes	Yes

Example

The following example looks up a main node based on an integer value.

```
int anyEntityId = 99;
mainNode = GwMainNodeLookup( anyEntityId );
```

Example

The following example looks up a main node based on a specified IP address.

```
Record mainNode = GwMainNodeLookup( "9.196.131.49" );
```

Example

The following example looks up a main node based on a system name.

```
text sysName = "fred";
mainNode = GwMainNodeLookup( sysName );
```

GwMainNodeLookupUsing()
The `GwMainNodeLookupUsing();` rule performs the same operation as the `GwMainNodeLookup()` rule, except that a field within the current top-level in-scope event is used to perform the lookup.

Syntax

The `GwMainNodeLookupUsing();` statement uses following syntax.

```
GwMainNodeLookup ( event field name);
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 61. Arguments of GwMainNodeLookupUsing()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`event field name`	Extract the value of this field from the current event, and use it to look for the main node in the topology.	Yes	Yes	Yes

Example

The following examples look up a main node based on a field within the current top-level in-scope event.

```
Record mainNode = GwMainNodeLookupUsing( "LocalNodeAlias" );
```

Example

The following examples look up a main node based on a field within the current top-level in-scope event.

```
text sysName = "MyCustomSysNameField";
mainNode = GwMainNodeLookup( sysName );
```

Example

The following examples look up a main node based on a field within the current top-level in-scope event.

```
mainNode = GwMainNodeLookup( "NmosEntityId" );
```

GwManagedStatus()
Retrieves the managed status of a specified entity. This rule checks for managed status by containment, and returns the actual managed status of the entity. For example, if the entity ID represents an entity contained in an unmanaged main node, then the contained entity is implicitly unmanaged and this rule returns the status as unmanaged, regardless of the status given in the ncimCache.managedStatus table.

Syntax

The `GwManagedStatus();` statement uses the following syntax.

```
GwManagedStatus ( entity ID );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 62. Arguments of GwManagedStatus()				
Argument	**Description**	**Accepts constants**	**Accepts variables**	**Accepts eval clauses**
`entity ID`	Integer entity ID of the entity.	Yes	Yes	Yes

Returns

This stitcher returns the managed status value for the specified entity.

Updating the current in-scope event

This example retrieves the managed status of the entity with entity ID 99. If this entity is listed as unmanaged in the managedStatus table, or a containing main node of entity ID 99 is unmanaged, then this rule will return the managed status value for entity ID 99 as unmanaged.

```
int nmosEntityId = 99;
int status = GwEntityData( nmosEntityId );
```

GwPipeComposition()
The `GwPipeComposition();` rule retrieves a list of pipe compositions for a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.pipeComposition`.

Syntax

The `GwPipeComposition();` statement uses following syntax.

```
GwPipeComposition ( entityId or entityName);
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 63. Arguments of GwPipeComposition()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return a list of pipe compositions for this entity.	Yes	Yes	Yes

Example

The following example finds pipe compositions for a specified entity and then prints out all of the results.

```
int entityId = eval( int, '&NmosEntityId' );

# Find all pipe compositions for the given entityId...
RecordList pipeCompositions = GwPipeComposition( entityId );

# ...and iterate through the returned results
Record pipeCompositionRow;
foreach ( pipeCompositions )
{
         pipeCompositionRow = GetInScopeRecord();
         PrintRecord( pipeCompositionRow );
}
```

Returns

This rule returns a list of records. Each record contains information about a single pipe composition, as contained in the `ncimCache.pipeComposition` table.

The following snippet shows an example of the results returned by this rule.

```
{
        ENTITYNAME='pe6-cr38.core.eu.test.lab[ Gi0/1 ]_
        p4-cr28.core.eu.test.lab[ 0 [ 2 ] ]';
        AGGREGATIONSEQUENCE=1;
},
{
        ENTITYNAME='p4-cr28.core.eu.test.lab[ Se0/0/1:0.202 ]_
        pe7-cr38.core.eu.test.lab[ Se0/0/0:0.202 ]';
        AGGREGATIONSEQUENCE=2;
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

GwProtocolEndPoint()
The `GwProtocolEndPoint();` rule retrieves a list of entities directly connected to a specified entity. This data is retrieved from the NCIM cache table, `ncimCache.protocolEndPoint`.

Syntax

The `GwProtocolEndPoint();` statement uses following syntax.

```
GwProtocolEndPoint ( entityId or entityName );
```

Arguments

The following table lists the properties of the arguments of this stitcher rule.

Table 64. Arguments of GwProtocolEndPoint()

Argument	Description	Accepts constants	Accepts variables	Accepts eval clauses
`entityId` or `entityName`	Instructs the rule to return a list of end points on this entity.	Yes	Yes	Yes

Example

The following example retrieve a list of end points on a specified entity and then prints out all of the results.

```
text entityName = eval( int, '&EntityName' );

# Find all end points on the given entityName...
RecordList endPoints = GwProtocolEndPoint( entityName );

# ...and iterate through the returned results
Record protocolEndPointRow;
foreach ( endPoints )
{
         protocolEndPointRow = GetInScopeRecord();
         PrintRecord( protocolEndPointRow );
}
```

Returns

This rule returns a list of records. Each record contains information about a single end point, as contained in the `ncimCache.protocolEndPoint` table.

The following snippet shows an example of the results returned by this rule.

```
{
        ENTITYNAME='Area_0.0.0.2_OSPF_ProtocolEndPoint_172.20.98.51_RD_[1]';
},
{
        ENTITYNAME='PIM_ENDPOINT_glasgow-gw-cr26.uk.eu.test.lab[ Fa0/0.1 ]';
},
{
        ENTITYNAME='IPMRoute_ProtocolEndPoint_(0.0.0.0, 224.0.1.39)_
        glasgow-gw-cr26.uk.eu.test.lab[ Fa0/0.1 ](downstream)';
},
{
        ENTITYNAME='IGMP_ENDPOINT_glasgow-gw-cr26.uk.eu.test.lab[ Fa0/0.1 ]';
},
{
        ENTITYNAME='glasgow-gw-cr26.uk.eu.test.lab[ Fa0/0.1 ] IP: 172.20.98.51';
```

```
},
{
        ENTITYNAME='glasgow-gw-cr26.uk.eu.test.lab[ Fa0/0.1 ]
        IP: 2001:15f8:106:212::51';
}
```

For more information on the NCIM cache tables see the *IBM Tivoli Network Manager Reference*.

Related reference

ncimCache database

This database stores topology updates from DNCIM.

Root-cause analysis stitcher rules

Use these stitcher rules for reference when you are working with root-cause analysis (RCA) stitchers.

Standard RCA Stitcher rules

These stitcher rules are used in standard root-cause analysis processing.

All of the standard RCA Stitcher rules operate with reference to the in-scope trigger event, and this event is referenced using the value in its `Serial` field. These stitcher rules work with a copy of the trigger event and this copy of the event is not actually updated during processing. Following processing the copy of trigger event simply goes out of scope without being updated. All updates are made to the original trigger event held in the RCA plug-in `mojo.events` table.

AmosDeleteEvent()

The `AmosDeleteEvent();` rule removes the trigger event from mojo.events.

Syntax

The `AmosDeleteEvent();` statement uses following syntax.

```
AmosDeleteEvent ();
```

Arguments

This stitcher rule takes no arguments.

Example

The following example shows how this rule is used in the `ProcessResolutionEvent.stch` stitcher.

```
int retval = 0
retval = AmosDeleteEvent();
```

This is equivalent to the following OQL action.

```
delete from mojo.events where Serial=Serial;
```

Where *Serial* is the serial number of the in-scope event at `nestlevel` 0.

AmosReprocessSuppressees()

The `AmosReprocessSuppressees();` rule reprocesses all suppressed events of the cleared or deleted trigger events.

Syntax

The `AmosReprocessSuppressees();` statement uses following syntax.

```
AmosReprocessSuppressees (  );
```

Arguments

This rule does not take any arguments.

Example

The following example reprocesses all suppressed events.

```
int numberOfSuppressees = 0;
numberOfSuppressees = AmosReprocessSuppressees();
```

AmosSetCause()

The `AmosSetCause();` rule makes the trigger event a root cause or unknown cause event. In either case, the event is not suppressed.

Syntax

The `AmosSetCause();` statement uses following syntax.

```
AmosSetCause ( causeType );
```

Example

The following example sets the cause type of an event.

```
int causeType = eval(int, '$RCA_ROOT_CAUSE');
success = AmosSetCause(causeType)
```

AmosSuppressByPeer()

The `AmosSuppressByPeer();` rule suppresses the trigger event with an existing event from mojo.events that is on a remote peer or remote neighbour. This is relevant for certain BGP and OSPF events.

Syntax

The `AmosSuppressByPeer();` statement uses following syntax.

```
AmosSuppressByPeer ( remoteNodeEntityId );
```

Example

The following example retrieves a remote node entity ID for the trigger event and then attempts to suppress the trigger event.

```
int suppressionType = eval(int, '$RCA_NO_SUPPRESSION');

// Get the RemoteNodeEntityId in this trigger event
int remoteNodeEntityId=0;
remoteNodeEntityId = eval(int,'&RemoteNodeEntityId');

if (remoteNodeEntityId > 0)
{
    suppressionType = AmosSuppressByPeer(remoteNodeEntityId);
}
```

AmosSuppressEvents()
The `AmosSuppressEvents();` rule is called by each of the suppression rules, such as ContainedEntitySuppression.stch, in order to suppress events, using a specified suppression type.

Syntax

The `AmosSuppressEvents();` statement uses following syntax.

```
AmosSuppressEvents ( suppressionType );
```

Example

The following example performs event suppression.

```
int suppressionType = eval(int, '$RCA_CONTAINED_SUPPRESSION');
int numberOfSuppressedEvents = 0;
numberOfSuppressedEvents = AmosSuppressEvents(suppressionType);
```

AmosSuppressTrigger()
The `AmosSuppressTrigger();` rule suppresses the trigger event with an existing event from the `mojo.events` table.

Syntax

The `AmosSuppressTrigger();` statement uses following syntax.

```
AmosSuppressTrigger (  );
```

Example

The following example shows how this rule is used to return the suppression type if the trigger event was suppressed. The value 0 means that the trigger event was not suppressed.

```
int suppressionType = eval(int, '$RCA_CONTAINED_SUPPRESSION');
suppressionType = AmosSuppressTrigger()
```

AmosTimedEventSuppression()
The `AmosTimedEventSuppression();` rule processes all events in the mojo.event table that have the `TimedEscalation` field set to 1 and that were created at least a specified number of seconds ago. The rule returns the number of events that were processed.

Syntax

The `AmosTimedEventSuppression();` statement uses following syntax.

```
AmosTimedEventSuppression ( age );
```

Example

The following example processes all events in the mojo.event table that have the `TimedEscalation` field set to 1 and that were created at least 30 seconds ago. The rule returns the number of events that were processed.

```
// Specify that the event's CreateTime  value must be at least 30 seconds ago
int age = 30;   // Specifies time in seconds
int numEventsProcessed = 0;
numEventsProcessed = AmosTimedEventSuppression(age);
```

RCA stitcher rules for customization
Use these stitchers to retrieve extra data to use in root-cause analysis processing. These stitcher rules are not used in standard root-cause analysis processing should only be used by advanced users.

AmosGetConnectedEntities()
The `AmosGetConnectedEntities();` rule retrieves all entities connected to an entity with a specified `entityId` and `entityType`.

Syntax

The `AmosGetConnectedEntities();` statement uses following syntax.

```
AmosGetConnectedEntities ( entityId, entityType );
```

Example

The following example shows how to use the `AmosGetConnectedEntities();` rule.

```
int entityId=0;
entityId = eval(int,'&NmosEntityId');

int entityType=0;
entityType = eval(int,'&EntityType');

RecordList myEntities = NULL;
myEntities = AmosGetConnectedEntities(entityId, entityType);
```

AmosGetContainedEntities()
The `AmosGetContainedEntities();` rule retrieves all entities contained by the entity with a specified `entityId`.

Syntax

The `AmosGetContainedEntities();` statement uses following syntax.

```
AmosGetContainedEntities ( entityId );
```

Example

The following example shows how to use the `AmosGetContainedEntities();` rule.

```
int entityId=0;
entityId = eval(int,'&NmosEntityId');

RecordList myEntities = NULL;
myEntities = AmosGetContainedEntities(entityId);
```

AmosGetContainerEntities()
The `AmosGetContainerEntities();` rule recursively retrieves all entities containing an entity with a specified `entityId`.

Syntax

The `AmosGetContainedEntities();` statement uses following syntax.

```
AmosGetContainerEntities ( entityId );
```

Example

The following example shows how to use the `AmosGetContainerEntities();` rule.

```
int entityId=0;
entityId = eval(int,'&NmosEntityId');

RecordList myEntities = NULL;
myEntities = AmosGetContainerEntities(entityId);
```

AmosGetEvents()
The `AmosGetEvents();` rule retrieves all events on a specified `entityId`.

Syntax

The `AmosGetEvents();` statement uses following syntax.

```
AmosGetEvents ( entityId );
```

Example

The following example shows how to use the `AmosGetEvents();` rule.

```
RecordList myEvents = NULL;
myEvents = AmosGetEvents(myEntityId);
```

AmosGetIsolatedEntities()
The `AmosGetIsolatedEntities();` rule retrieves all entities isolated by the entity with the specified `entityId` value with respect to the specified poller entity.

Syntax

The `AmosGetIsolatedEntities();` statement uses following syntax.

```
AmosGetIsolatedEntities ( entityId, pollerEntityId );
```

Example

The following example shows how to use the `AmosGetIsolatedEntities();` rule.

```
int entityId=0;
entityId = eval(int,'&NmosEntityId');

int pollerEntityId=0;
pollerEntityId = eval(int,'&PollerEntityId');

RecordList myEntities = NULL;
myEntities = AmosGetIsolatedEntities(entityId, pollerEntityId);
```

AmosRetrieveEntities()
The `AmosRetrieveEntities();` rule retrieves entities relative to the trigger entity; for example, it can retrieve all entities contained by the trigger entity.

Syntax

The `AmosRetrieveEntities();` statement uses following syntax.

```
AmosRetrieveEntities ( suppressionType );
```

Example

The following example shows how to use the `AmosRetrieveEntities();` rule.

```
int suppressionType = eval(int, '$RCA_CONTAINED_SUPPRESSION');

RecordList myEntities = NULL;
myEntities = AmosRetrieveEntities(suppressionType);
```

AmosUpdateEvent()
The `AmosUpdateEvent();` rule updates the event in the table, `mojo.events`. The rule uses the Serial number of the in-scope event to do this. The rule does not update the in-scope record itself. The in-scope records simply goes out of scope at the end of the operation.

Syntax

The `AmosUpdateEvent();` statement uses following syntax.

```
AmosUpdateEvent ( attribute nameattribute value );
```

Example

The following example shows how to use the `AmosUpdateEvent();` rule to update an integer field.

```
int success = 0;
success = AmosUpdateEvent("NmosCauseType", 1);
```

This is equivalent to the following OQL action:

```
update mojo.events set NmosCauseType=1 where Serial=Serial;
```

Where *Serial* is the serial number of the in-scope event.

Example

The following example shows how to use the `AmosUpdateEvent();` rule to update text string fields.

```
success = AmosUpdateEvent("NmosSerial", "0");
success = AmosUpdateEvent("SuppressionState", "0");
success = AmosUpdateEvent("SuppressionTime", "0");
```

Example

The following is a further example of how to use the `AmosUpdateEvent();` rule to update an integer field.

```
int success = 0
text myname = "NmosCauseType";
int  myvalue = 2;
success = AmosUpdateEvent(myname, myvalue);
```

AmosUpdateSuppressees()
The `AmosUpdateSuppressees();` rule assigns a value to one field in all of the suppressed events and then sets this flag to a specified value.

Syntax

The `AmosUpdateSuppressees();` statement uses following syntax.

```
AmosUpdateSuppressees ( attribute name, attribute value );
```

Example

The following example finds all the events currently suppressed by the trigger event, and flags them as orphans because the trigger event has just been cleared or deleted. The orphan events are subsequently reprocessed by the AmosReprocessSuppressees rule called later in the stitcher. The parameters set the field `Orphaned` to the value `1` in each of the suppressee events.

```
int numberOfSuppressees = 0;
numberOfSuppressees = AmosUpdateSuppressees("Orphaned", 1);
```

Stitcher language building blocks

To construct statements in the stitcher language, use the programming constructs, relational operators, and datatypes.

Programming constructs

The following table lists the stitcher language programming constructs.

Table 65. Stitcher language programming constructs

Stitcher keyword	Description
delete	Deletes lists created by the RetrieveOQL function.
else	Executes statements where the conditional test specified with the `if` loop is not passed.
for	Defines a loop that is repeated a specified number of times.
foreach	Performs an action on each member of a list of datatype `RecordList`.
if	Executes statements if a conditional test is passed.
while	Defines a loop that repeats while a condition holds true.

Datatypes

The following table describes the stitcher language datatypes.

Table 66. Stitcher language datatypes

Datatype	Description
int	Stores integer values.
Record	Stores a single returned record.
RecordList	Stores lists created by the `RetrieveOQL` function.
text	Stores string values.

Relational operators

The following table lists the relational operators.

Table 67. Stitcher language relational operators

Operator	Description
==	Test for equality.
!=	Test for non-equality.
<	Test for less than.

Table 67. Stitcher language relational operators (continued)	
Operator	**Description**
>	Test for greater than.
<=	Test for less than or equal to.
>=	Test for greater than or equal to.

Related reference

Stitcher rules
The stitcher rules are specified within the `StitcherRules{}` section of a stitcher. Stitcher rules determine how a stitcher functions.

Quotes in OQL
In OQL the TEXT datatype must be enclosed by *matching* quotation marks (either single or double quotes).

delete()
The `delete` rule removes lists and records that have been created and are no longer needed.

The for loop
The `for` loop is used to repeat a set of rules a given number of times.

The while Loop
The `while` loop is used to execute a series of instructions while a specified condition remains true.

The if statement
The `if` statement performs an action if a particular condition is satisfied.

The foreach Loop
The `foreach` loop performs an action on every record stored in a variable of type `RecordList`.

Stitcher language comments

Comments are introduced by `--` or //. If a comment requires a carriage return, the characters on the next line must also be commented out.

The following example shows how to use comments.

```
-- This is a valid comment.
// This is also a valid comment.
```

Related reference

Features of OQL
The following topics describe the features of Object Query Language (OQL).

Precedence and association of operators

The rules for precedence and association of operators determine the grouping of operators with operands, and indicate the order in which the operators in an expression are executed.

For complex expressions, use parentheses to avoid ambiguity.

The following table describes the operators.

Table 68. OQL operators in order of decreasing precedence			
Operator	**Description**	**Associativity**	**Precedence**
-	Negative sign	Non-associative	1 (highest)
*	Multiplication	Left	2
/	Division	Left	2

Table 68. OQL operators in order of decreasing precedence (continued)			
Operator	**Description**	**Associativity**	**Precedence**
OR	Logical OR	Left	3
AND	Logical AND	Left	4
NOT	Logical NOT	Left	5
=	Equal to	Left	6
<>	Not equal to	Left	6
<	Less than	Left	6
>	Greater than	Left	6
<=	Less than or equal to	Left	6
>=	Greater than or equal to	Left	6
+	Addition	Left	7
-	Subtraction	Left	7 (lowest)

OQL quotes in the stitcher language

OQL statements embedded within the stitcher language (for example, within the `ExecuteOQL();` statement) are enclosed in double quotes.

If quotes are needed *inside* the embedded OQL, they must be single quotes even if double quotes would normally be used. The following example shows an embedded OQL statement that is enclosed in double quotes. The TEXT datatype within the statement is enclosed in single quotes:

```
ExecuteOQL(
        "select m_Name from finders.returns where m_Creator = 'PingFinder';"
            );
```

Elsewhere in the stitcher language, single quotes are generally used, as shown in the following example:

```
ExecuteStitcher( 'CreateAndSendTopology' );
```

Domain-specific stitchers

If you want to make custom stitching easier to debug, or you want to keep the original stitcher files unaltered, you can create domain-specific stitchers.

Advantages and operation of domain-specific stitchers

To make a stitcher domain-specific, save a copy of it with the domain name appended to the filename. For example: `LinkDomainsLoadPresetConnections.DOMAIN1.stch`.

Domain-specific stitchers only process connections that start in the domain defined in their filename.

If you have a large number of connections that you want to create, having multiple stitcher files can reduce the size and complexity of individual files.

Chapter 3. Syntax for poll definition expressions

Use this information to understand how to build complex threshold expressions to use in basic and generic threshold poll definitions.

eval statement syntax in threshold expressions

Use this information to understand how to use the `eval` statement to create complex threshold expressions within basic and generic threshold poll definitions.

eval statement syntax for SNMP variables

You can evaluate SNMP variables using the `eval` statement.

The following examples illustrate how to evaluate SNMP variables using the `eval` statement.

Sample: Evaluation of SNMP values

The following example returns the value of the SNMP variable `sysName`.

```
eval(text, '&SNMP.VALUE.sysName')
```

Sample: Evaluation of SNMP indices

The following example returns the value of the index of the SNMP request for the variable `ipRouteNextHop`. In a table poll, this is evaluated for every index in the table list..

```
eval(text, '&SNMP.INDEX.ipRouteNextHop')
```

Sample: Evaluation of previously retrieved SNMP values

The following example returns the value of the SNMP variable `sysName`, which was retrieved when this poll was last run..

```
eval(text, '&SNMP.VALUE.OLD.sysName')
```

Sample: Evaluation of the results of an expression

The following example returns the results of an expression, such as the SNMP Bandwidth poll. The results are written directly within the alert description within the IBM Tivoli Netcool/OMNIbus or **Event Viewer**.

Note: This feature is only available in basic threshold polls. The feature is not available in generic threshold polls, because generic threshold polls do not evaluate to a result, they only evaluate to true or false. If you attempt to use this syntax in a generic threshold, the operation will not fail; however, it will generate a blank space in the alert.

```
eval(text,'&POLLDATA.RESULT')
```

Sample: Evaluation of Old SNMP Indices

The following example returns the value of the index of the SNMP request for the variable `ipRouteNextHop`, which was retrieved when this poll was last run. In a table poll, this is evaluated for every index in the table list. Note that the old index is likely to be the same as the new index.

```
eval(text, '&SNMP.INDEX.OLD.ipRouteNextHop')
```

eval statement syntax for network entity variables

You can evaluate network entity variables, such as the value of an entity ID or entity name, using the `eval` statement.

The `ENTITY` keyword can be used in threshold expressions or descriptions to evaluate the value of network entity variables. The following examples illustrate how to evaluate network entity variables using the `ENTITY` keyword in the `eval` statement.

Sample: Evaluation of the value of the entityName of the containing chassis

The following example can be used in a threshold expression or threshold description, and shows how to evaluate the value of the entityName corresponding to the chassis that contains the entity being monitored.

```
An interface on eval(text, '&ENTITY.MAINNODEENTITYNAME') is down
```

Attributes of the ENTITY keyword

Valid attributes of the `ENTITY` keyword are listed in the following table. For information on the NCIM database fields cited in the table, see the *IBM Tivoli Network Manager Reference*.

Table 69. Attributes of the ENTITY keyword

Attribute	Description
ENTITYID	Value of the field entity.entityId for the monitored entity. This can be either an interface or chassis.
ENTITYNAME	Value of the field entity.entityName for the monitored entity. This can be either an interface or chassis.
ENTITYTYPE	Value of the field entity.entityType for the monitored entity. This can be either an interface or chassis.
ENTITYCLASS	Value of the field entity.className for the monitored entity. This can be either an interface or chassis.
ACCESSIPADDRESS	Value of the field interface.accessIPAddress or chassis.accessIPAddress, depending on what type of poll it is.
IFINDEX	For interface polls, the value of the field interface.ifIndex for the monitored entity.
IFTYPESTRING	For interface polls, the value of the field interface.ifTypeString for the monitored entity
IFNAME	For interface polls, the value of the field interface.ifName for the monitored entity.
IFDESCR	For interface polls, the value of the field interface.ifDescr for the monitored entity.
IFALIAS	For interface polls, the value of the field interface.ifAlias for the monitored entity.
INSTANCESTR	For interface polls, a string representation of the field interface.instanceStr for the monitored entity.
ENTITYMANAGED	Indicates whether the monitored entity is in a managed state, determined by whether the field managedStatus.status is either not present or zero for the entity in question.

Table 69. Attributes of the ENTITY keyword (continued)	
Attribute	**Description**
CHASSISMANAGED	Indicates whether the chassis containing the entity being monitored is in a managed status, determined by whether the field managedStatus.status is either not present or zero for the chassis in question.
MAINNODEADDRESS	Value of accessIPAddress for the chassis containing the entity being monitored
MAINNODEENTITYNAME	Value of the field entityName for the entity record corresponding to the chassis containing the entity being monitored.
MAINNODEENTITYID	Value of the field chassis.entityId for the chassis containing the entity being monitored

eval statement syntax for poll policy variables

You can evaluate poll policy variables, such as the name of a policy or the ID of the domain in which this poll policy is found, using the `eval` statement.

The POLICY keyword can be used in threshold expressions or descriptions to evaluate the value of poll policy variables. The following examples illustrate how to evaluate poll policy variables using the POLICY keyword in the `eval` statement.

Sample: Evaluation of the value of poll policy name and related domain ID

The following example can be used in a threshold expression or threshold description, and shows how to evaluate the value of name of the poll policy and the ID of the domain in which this poll policy is found.

```
The eval(text, '&POLICY.POLICYNAME') policy polls entities in
domain number eval(text, '&POLICY.DOMAINMGRID)
```

Attributes of the POLICY keyword

Valid attributes of the POLICY keyword are listed in the following table. For information on the NCIM database fields cited in the table, see the *IBM Tivoli Network Manager Reference*.

Table 70. Attributes of the POLICY keyword	
Attribute	**Description**
POLICYID	Unique integer identifier for this poll policy.
DOMAINMGRID	Foreign key referencing the NCIM domainMgr table. Specifies the ID for the domain of the monitored entity.
POLICYNAME	Name of this poll policy.

eval statement syntax for poll definition variables

You can evaluate poll definition variables, such as the name of a poll definition or the severity of failure events raised by policies using a poll definition, using the `eval` statement.

The POLL keyword can be used in threshold expressions or descriptions to evaluate the value of poll definition variables. The following examples illustrate how to evaluate poll definition variables using the POLL keyword in the `eval` statement.

Sample: Evaluation of the value of poll definition name and associated event severity

The following example can be used in a threshold expression or threshold description, and shows how to evaluate the value of name of the poll definition and the severity of the events generated by poll policies that use this poll definition.

```
Poll policies that use the eval(text, '&POLL.TEMPLATENAME') poll definition
generate events with severity eval(text, '&POLL.EVENTSEVERITY')
```

Attributes of the POLL keyword

Valid attributes of the POLL keyword are listed in the following table.

Table 71. Attributes of the POLL keyword

Attribute	Description
TEMPLATEID	Unique identifier for this poll definition.
TEMPLATENAME	Name of this poll definition.
TEMPLATETYPE	Type of poll definition. This value is derived from the list the user is presented with when creating a new poll definition.
EVENTNAME	Text identifier to be used for events raised by poll policies that use this poll definition. This text is written to the alerts.status table as the EventId field, unless the text is modified by the rules file of the probe for Tivoli Netcool/OMNIbus (nco_p_ncpmonitor).
EVENTSEVERITY	Severity of failure events raised by poll policies using this poll definition. This text is written to the alerts.status table as the Severity field, unless the text is modified by the rules file of the probe for Tivoli Netcool/OMNIbus (nco_p_ncpmonitor).
POLLINTERVAL	Interval in seconds at which each entity in scope for this poll is polled.

Operators in threshold expressions

Use this information to understand which operators to use to create threshold expressions in basic and generic threshold poll definitions.

The following table lists the operators that you can use in threshold expressions.

Table 72. Operators in threshold expressions

Operator	Example
Plus	(1 + 2)
Minus	(4 - 2)
Multiplication	(5 * 3)
Division	(10 / 2)
Modulus	(8 % 3)
Power®	(10 POW 3)
Log	(Ln 5)
IP to Long datatype conversion	(IpToLong("1.2.3.4"))
Bitwise AND	(5 & 3) **Note:** Bitwise operations can only be applied to integer values.

Table 72. Operators in threshold expressions (continued)	
Operator	**Example**
Bitwise	(5 | 3)
Bitwise Exclusive OR	(5 ^ 3)
Boolean OR	((eval(int, '&SNMP.VALUE.ifSpeed') > 10000) OR (eval(int, '&SNMP.VALUE.ifSpeed') < 100))
Boolean AND	((eval(int, '&SNMP.VALUE.ifSpeed') > 10000) AND (eval(int, '&SNMP.VALUE.ifOperStatus') !=2))
Boolean NOT	(NOT((eval(int,'&SNMP.VALUE.ifOperStatus') = 1))
Equal	(eval(int, '&SNMP.VALUE.ifOperStatus') = 1)
Not equal	(eval(int, '&SNMP.VALUE.ifOperStatus') != 1)
Less than	(eval(int, '&SNMP.VALUE.ifSpeed') < 100)
Greater than	(eval(int, '&SNMP.VALUE.ifSpeed') >100)
Less than or equal	(eval(int, '&SNMP.VALUE.ifSpeed') <= 100)
Greater than or equal	(eval(int, '&SNMP.VALUE.ifSpeed') >= 100)
Like	(eval(text, '&ENTITY.IFDESCR') LIKE 'Gigabit.*')
Not Like	(eval(text, '&ENTITY.IFDESCR') NOT LIKE 'Loopback.*')

Chapter 4. Active Object Class files

Active Object Class (AOC) files are used to define the device class hierarchy upon which Network Manager automatically classifies all discovered network devices after the completion of a discovery. By default, AOC files are stored in the `NCHOME/precision/aoc` directory.

The following topics describe AOC syntax.

Device class hierarchy

Network Manager uses a class hierarchy to model network devices.

The following figure shows an example class hierarchy.

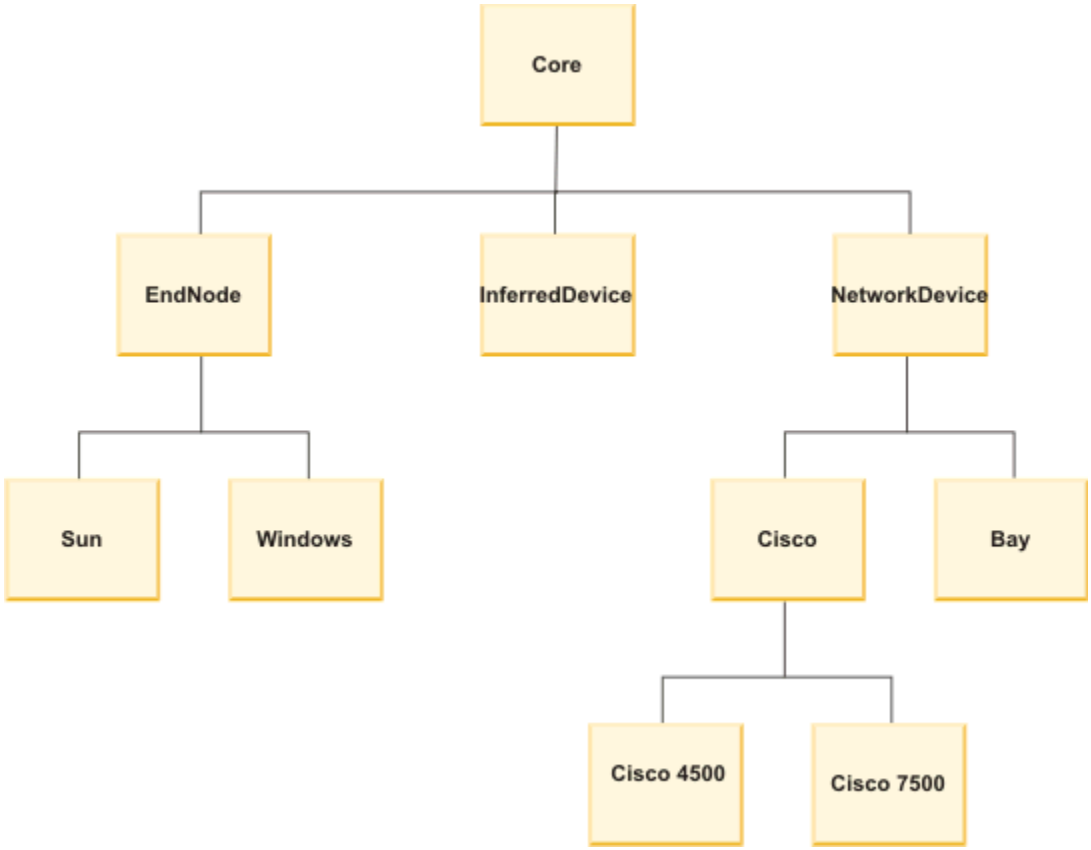


Figure 1. Class hierarchy of network devices

The management policies specified in the class hierarchy describe how to handle instances of devices and how to handle events related to them. The class hierarchy is available for polling operations. You can specify which classes to include in each polling operation through the poll policy settings.

Network Manager does not permit multiple inheritance, which is the ability of a new class to inherit the characteristics from more than one class.

The following topics describe the classes contained in the hierarchy.

EndNode class

The EndNode class contains devices such as workstations and printers.

Within the EndNode class, the NoSNMPAccess class contains devices that were discovered and added to the topology even though Network Manager had no SNMP access to them, such as workstations. Devices might not have SNMP access for one of the following reasons:

- The device does not have SNMP capabilities.
- Network Manager does not have the SNMP credentials to access the device.

InferredDevice class

The InferredDevice class contains devices that Network Manager could not discover directly, although they are considered to exist on the network.

You might infer the existence of CE routers for your customer, for example, by specifying the CE routers in the advanced discovery configuration options.

NetworkDevice class

The NetworkDevice class contains all device types grouped into subclasses according to their manufacturer.

All Cisco devices, for example, are contained within the Cisco subclass. The default poll policies are usually defined for network devices and their derived classes.

AOC syntax

Use this information to understand the structure of AOC files.

In addition to the information in this topic, you should also read the OQL language syntax information, in particular, the eval statement.

The syntax and naming conventions described in the following table are used throughout the AOCs.

Table 73. Syntax characters used in AOCs

Character	Name	Meaning
[ ]	Square brackets	Typically defines a list of items. There can be zero or more items within the brackets and each item must be separated by a single comma.
{ }	Curly brackets	Typically defines an object.
" "	Double quotes	Typically used to enclose assignments to various attributes (of datatype text) within an AOC. As a general rule all assignments use double quotes.
' '	Single quotes	Typically used to enclose evaluations of column names or system variables within an eval statement. Single quotes can be used within double quotes, but double quotes cannot be used within single quotes.
,	Single comma	Typically used either as a separator between elements of a list or to terminate an assignment to a particular attribute.
;	Semi colon	Typically used to terminate assignments to the major components listed below.

Components of an AOC file

Use this generic description of the structure of an AOC to help you write new AOC files and understand existing AOC files.

AOC components

The major components of the AOC and descriptions of the components are provided below.

- Name of the active object class (active object)
- Super class (super_class)
- Instantiate rule (instantiate_rule)
- Icon used in GUI to represent the device class

Name

In the active object attribute, you declare the name of the current class.

You can specify any unique text string between the double quotes. The entire AOC file is contained within the first pair of curly brackets. A semicolon follows the final bracket to terminate the AOC.

```
active object 'Linux'
{
        super_class = 'EndNode';

          instantiate_rule = "EntityOID = '1.3.6.1.4.1.2021.250.10' OR
              EntityOID = '1.3.6.1.4.1.8072.3.2.10' OR
              EntityOID = '1.3.6.1.4.1.1575.1.5'";

          visual_icon = 'EndNode';
};
```

Super class

You can configure the Super class to define the name of the AOC from which the current class inherits.

The name between the double quotes must be the name of an already-defined AOC. In the AOC hierarchy, Core is the only class that has an unassigned super_class. When editing any other class, the super_class must never be left empty.

```
super_class = "Core";
```

Instantiate rule

You code the instantiate rule as a logical test against the attributes of the entity. The most specific class that matches the test defines the class the object belongs to. The test is done by first testing the Core.aoc class and then its subclasses in turn.

If an object meets the instantiation criteria for more than one class, it automatically instantiates to the lowest leftmost class in the hierarchy. The following example shows a specification of the rule that instantiates everything by default.

```
instantiate_rule = "EntityOID like '.*'";
```

Visual icon

You can assign icons to device classes. These icons represent devices of that class in the network visuulization GUIs, including the Network Views and the Network Hop View.

For information on how to assign icons to device classes, see the *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*.

Part 2. Perl API reference

Read about the Perl API provided with Network Manager.

Chapter 5. Overview of the Perl API

The Perl API provides developers with the functionality to write discovery agents and other client/server applications. These applications can perform such tasks as accessing and modifying records in Network Manager databases, and retrieving SNMP information from a network device. Developers can also integrate third-party products using the Perl API as a tool to interface with Network Manager.

RIV module overview

The RIV module provides a variable, functions, and virtual methods that the Perl API application modules — `RIV::Agent` and `RIV::App` — use.

Perl API modules used with the RIV module

The following table identifies and briefly describes the Perl API modules used with the `RIV` module:

Perl API Module	Description
`RIV::Agent`	Provides an interface for implementing Network Manager discovery agents.
`RIV::App`	Provides an interface for implementing other Network Manager client/server applications.
`RIV::OQL`	Provides an interface to communicate and perform operations on internal Network Manager databases.
`RIV::Param`	Provides an interface for parsing standard and Network Manager application-specific command line arguments.
`RIV::Record`	Provides a data structure to store the network entity. Typically, you use this data structure in conjunction with the `RIV::Agent` module to write discovery agents.
`RIV::RecordCache`	Provides an interface to access records that reside in a cache.
`RIV::SnmpAccess`	Provides an interface to perform SNMP-related operations on Network Manager MIB trees. **Note:** Discovery agents in previous versions of the Perl API used this module to obtain SNMP information from network devices. Discovery agents implemented with this version of the Perl API should use the SNMP methods that the `RIV::Agent` module provides.

Types of applications

There are two types of applications that you can write using the Perl API:

- Discovery agents — Use the `RIV::Agent` constructor and the ncp_disco_perl_agent binary to create discovery agent applications.
- Other client/server applications — Use the `RIV::App` constructor and the ncp_perl binary to other client/server applications. Examples of these other client/server applications include those that access Network Manager databases.

These application objects are required for interaction with Network Manager components (through the virtual methods exported through the `RIV` module) and for instantiation of the other `RIV` modules.

Application objects that the RIV::Agent and RIV::App constructors return are identical for the purpose of accessing other module functionality (for example, RIV::OQL).

RIV module functions

The following table identifies and briefly describes the functions that the RIV module provides for Network Manager discovery agents and other Network Manager client/server applications:

RIV module function	Description
RIV::GetInput	This function has been deprecated. Use the RIV::GetResult function.
RIV::GetResult	Obtains input either directly or indirectly from message broker.
RIV::InputFilter	Binds the specified input function to input tags that match the specified regular expression.
RIV::InputQueueLength	Returns the number of items waiting in the application's input queue.
RIV::IsIpNotLoopBackOrMulticast	Determines whether the specified address is a valid IP address and not a loop back or multicast address.
RIV::IsIpValid	Determines whether the specified address is a valid IP address.
RIV::IsIpv4Valid	Determines whether the specified address is a valid IPv4 address.
RIV::IsIpv6Valid	Determines whether the specified address is a valid IPv6 address.
RIV::ReadDir	Returns a reference to an array of filenames contained in the specified directory.
RIV::RivDebug	Prints a list of debug message strings to the standard output.
RIV::RivMessage	Prints a list of log message strings to the standard output.
RIV::RivError	Displays error messages.

See "RIV module reference" on page 165 for the reference (man) pages associated with these functions.

RIV module virtual methods

The following table identifies and briefly describes the virtual methods that the RIV module provides for Network Manager discovery agents and other Network Manager client/server applications:

RIV module virtual method	Description
AddSubject	Binds the application to the specified message broker subject.
AddTimer	Creates a single-shot or repeating timer.

RIV module virtual method	Description
DebugLevel	Provides access to the global Network Manager debug setting through the RIV::DebugLevel variable.
DecryptPassword	Decrypts a password that was previously encrypted in a previous call to the EncryptPassword RIV module virtual method.
EncryptPassword	Returns an encrypted representation of the specified password.
Latency	Retrieves the timeout for queries.
PostInput	Adds a message to the queue.
PublishMessage	Publishes the specified message string.
PublishMessage	Encodes the hash reference into a message broker string.
RetryLimit	Sets the retry limit for queries or returns the maximum number of retries for queries.

See "RIV module reference" on page 165 for the reference (man) pages associated with these functions.

RIV::Agent module overview

The RIV::Agent module provides an interface for implementing Network Manager discovery agents. A discovery agent is a specialized application that retrieves connectivity-related information for network entities.

RIV::Agent constructor

The RIV::Agent module provides a constructor that creates a discovery agent application object. Use this application object to:

- Interact with Network Manager core components libraries using the virtual methods exported from the RIV module.
- Instantiate objects for and interact with the other Perl modules: RIV::Param, RIV::Record, and RIV::RecordCache.

Input data records

Input data records that the discovery service sends are supplied through the RIV::GetResult method. These input data records can be stored as RIV::Record objects, which are nested hash lists to which you can add local and remote neighbors.

Note: All input data records that other services (including the OQL service) send are also supplied through the RIV::GetResult method.

Discovery agents and multiple threads

The RIV::Agent module allows you to implement discovery agents using multiple threads. The threads implementation creates a single master Perl interpreter that gets copied, one for each thread. Thus, if the discovery agent makes use of three threads, there will be three copies of the master interpreter. Specifically, the RIV::Agent module provides the LockThreads and UnLockThreads methods related to discovery agents and multiple threads.

SNMP operation methods

The `RIV::Agent` module provides methods that discovery agents use to obtain Simple Network Management Protocol (SNMP) information from network devices. These methods obtain this information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the SNMP-related methods can make the appropriate SNMP requests.

The following table identifies and briefly describes the SNMP operation methods that the `RIV::Agent` module provides:

SNMP method	Description
`SnmpGet`	Performs an SNMP `get` operation.
`SnmpGetNext`	Performs an SNMP `get-next` operation.
`SnmpGetBulk`	Performs an SNMP `get-bulk` operation.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

DNS operation methods

The `RIV::Agent` module provides methods that discovery agents use to obtain Domain Name System (DNS) information from network devices. These methods obtain this information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the DNS-related methods can make the appropriate DNS requests.

The following table identifies and briefly describes the DNS operation methods that the `RIV::Agent` module provides:

DNS method	Description
`GetDNSAllIpAddrs`	Gets all IP addresses corresponding to a particular node name.
`GetDNSAllNames`	Gets all node names corresponding to the specified IP addresses.
`GetDNSFirstIpAddr`	Gets the first IP address in the list of IP addresses for this node.
`GetDNSFirstName`	Gets the first node name in the list of node names for this IP address.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

Ping operation methods

The `RIV::Agent` module provides methods that discovery agents use to perform ping operations on network devices. Ping operations determine whether a specific IP or subnet address is accessible. Typically, the ping operation sends a packet to the specified address and waits for a reply.

The `RIV::Agent` module ping operation methods perform the specified ping operation through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that these ping-related methods can make the appropriate ping requests.

The following table identifies and briefly describes the ping operation methods that the `RIV::Agent` module provides:

ping method	Description
`GetPingIP`	Pings the specified IP address and returns whether a network device exists at that address.

ping method	Description
GetPingList	Pings the specified list of IP addresses and returns a list of network devices that exist at those addresses.
GetPingSubnet	Pings the specified subnet and returns whether one or more devices exist at that subnet.
Ping	Pings the specified IP address.
PingList	Pings the specified list of IP addresses.
PingSubnet	Pings the specified subnet.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

IP and MAC address operation methods

The RIV::Agent module provides methods that discovery agents use to perform operations on Internet Protocol (IP) and Medium Access Control (MAC) addresses. The RIV::Agent module IP and MAC address operation methods perform the specified address operation through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that these address-related methods can make the appropriate address operation requests.

The following table identifies and briefly describes the IP and MAC address operation methods that the RIV::Agent module provides:

Address method	Description
GetIpArp	Converts the specified MAC address to an IP address.
GetMacArp	Converts the specified IP address to a MAC address.
GetTraceRoute	Traces a route to the specified destination IP address and returns a list of network devices that reside on that route.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

Telnet operation methods

The RIV::Agent module provides methods that discovery agents use to obtain network device information through Telnet rather than SNMP. Like the SNMP methods, the Telnet methods obtain network device-related information through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the Telnet-related methods can make the appropriate Telnet requests.

The following table identifies and briefly describes the Telnet operation methods that the RIV::Agent module provides:

Telnet method	Description
GetMultTelnet	Executes multiple Telnet commands on the specified network device.
GetTelnet	Executes the specified Telnet command on the specified network device.
GetTelnetCols	Executes the specified Telnet command on the specified network device and splits the return data into table columns.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

Network entity operation methods

The RIV::Agent module provides methods that discovery agents use to perform operations on network entities. Specifically, the RIV::Agent module provides methods that perform the following network entity operations:

Network entity method	Description
SendNEToDisco	Sends processed records from RIV::Record to the returns table of the specified Agent database in Disco.
SendNEToNextPhase	Marks the network entity as having completed the current phase and puts the network entity back on the Agent queue ready for processing in the next phase.

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

Collector communication operation methods

The RIV::Agent module provides methods that discovery agents can use to call XML-RPC methods supported by Collectors. These XML-RPC Collector supported methods will return an XML response that needs to be parsed. The caller specifies either a custom XML-RPC method or one of the XML-RPC methods that the Network Manager Collector XML Schema defines. The Collector communication methods issue requests through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the Collector communication-related methods can make the appropriate XML-RPC requests.

The following table identifies and briefly describes the Collector communication operation methods that the RIV::Agent module provides to make the XML-RPC call:

XML-RPC method	Description
GetXMLRPCData	Issues an XML-RPC call, via the XML-RPC Helper, on the specified method to the specified host and port.
GetXMLRPCEntityData	Issues an XML-RPC call, via the XML-RPC Helper, on the specified method. The Collector contacted will be that referenced in the supplied standard entity record (that is, the .despatch record).

See "RIV::Agent module reference" on page 187 for the reference (man) pages associated with these methods.

RIV::App module overview

The RIV::App module provides an interface for implementing Network Manager client/server applications within one domain.

RIV::APP constructor

The RIV::App module provides two constructors that create a client/server application object. You use this client/server application object to:

- Interact with Network Manager core components libraries using the virtual methods exported from the RIV module.
- Instantiate objects for and interact with the other Perl modules: RIV::OQL, RIV::Param, RIV::Record, and RIV::RecordCache.

A client/server application can create one or more RIV::App application objects as required. For example, two instances of RIV::App application objects would be needed in order to implement some special purpose cross-domain behavior.

One example of a client/server application is one that performs one or more OQL queries (in which case the RIV::OQL module would also be used).

Note: The RIV::App module provides the interface for implementing all Network Manager client/server applications except for discovery agents. To write discovery agents, use the RIV::Agent module.

RIV::OQL module overview

The RIV::OQL module provides an interface to communicate with and perform operations on Network Manager internal databases.

RIV::OQL constructor

The RIV::OQL module provides a constructor that creates and initializes a new RIV::OQL object. The RIV::OQL constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the RIV::Agent or RIV::App constructor. The RIV::OQL constructor also takes the name of a service that indicates the Network Manager internal database to use.

Database operation methods

The RIV::OQL module provides methods that client/server applications use to perform a variety of operations on Network Manager internal databases.

The following table identifies and briefly describes the database operation methods that the RIV::OQL module provides:

Database method	Description
Close	Close an OQL client.
CreateDB	Creates a database.
CreateTable	Creates a database table.
Delete	Deletes records from a database table.
Insert	Inserts records into a database table.
Print	Prints records as a result of a database query.
Query	Run a query.
QueryGetResult	Get a single result.
QueryGetResults	Get multiple results.
Select	Executes the specified OQL statement.
Send	Communicates with the specified database.
Update	Updates records that currently reside in the specified database.

See "RIV::OQL module reference" on page 212 for the reference (man) pages associated with these methods.

RIV::Param module overview

The `RIV::Param` module provides an interface to parse standard and Network Manager application-specific command line arguments.

Standard arguments

Standard arguments are used to specify information about the Network Manager execution environment and to select debug output and application help. All Network Manager applications must support these arguments.

The standard arguments that the `RIV::Param` module provides are summarized in the following table:

Argument	Description
`-domain` *domain name*	Specifies the command line argument used to identify the domain in which a user wants to perform some task, for example, starting the Network Manager core components. The *domain name* argument specifies the name of the domain.
`-debug` *debug level*	Specifies the command line argument used to identify the level of debugging output that a user prefers. The *debug level* argument specifies a value from 1-4, where 4 represents the most detailed output.
`-latency` *query latency*	Specifies the command line argument used to specify the maximum time that CLASS waits to connect to another Network Manager process by means of the messaging bus. This option is useful for large and busy networks where the default settings can cause processes to assume that there is a problem when in fact the communication delay is a result of network traffic. The *query latency* argument specifies the maximum time in milliseconds (ms) that CLASS waits.
`-messagelevel` *message level*	Specifies the command line argument used to identify the level of message output that a user prefers. The *message level* argument specifies the level of message to be logged (the default is warn): • debug • info • warn • error • fatal
`-help`	Specifies the command line argument used to display command line options.

RIV::Param constructor

The `RIV::Param` module provides a constructor that creates and initializes a new `RIV::Param` object. The `RIV::Param` constructor takes three optional parameters used to specify:

- Application-specific parameters
- Usage information or nonstandard command line argument scenarios
- Help information written to standard output

The RIV::Param object can be used as the first parameter to the RIV::App constructor in place of the domain name argument. In this case, the application object is created with the specified values. Likewise, the RIV::Param object can be used as the first parameter to the RIV::Agent constructor.

See "RIV::Param Constructor" on page 225 for details.

RIV::Param module constants

The RIV::Param module also provides several constants that the RIV::Param constructor uses to identify a particular command line as follows:

Constant	Description
RivParamNoArg	Specifies that the command line takes no arguments.
RivParamSingleArg	Specifies that the command line takes one argument.
RivParamMandatory	Specifies that the command line takes a mandatory argument.

Parameter operation methods

The RIV::Param module provides methods that client/server applications use to print usage information or to obtain information about domain and command names.

The following table identifies and briefly describes the parameter operation methods that the RIV::Param module provides:

Parameter method	Description
CommandName	Returns the name of the command.
DomainName	Returns the name of the domain.
Usage	Writes a brief usage explanation to standard output.

See "RIV::Param module reference" on page 224 for the reference (man) pages associated with these methods.

RIV::Record module overview

The RIV::Record module provides a data structure to store network entity data records.

Network entities

The RIV::Record module is used in conjunction with the RIV::Agent module to write discovery agents. The RIV::Record data structure stores records associated with the network entities sent by DISCO to the Perl Discovery Agent. You can then add local neighbors and remote neighbors to this record by calling the appropriate local and remote neighbor operation methods.

RIV::Record constructor

Before accessing the methods that the RIV::Record module provides, you must call the RIV::Record constructor to create and initialize a new RIV::Record data structure. This data structure stores network entity records retrieved from DISCO.

RIV::Record data structure

The following respresents a RIV::Record data structure:

```
$refLocalNeighbours = $record->{m_LocalNbr};
@LocalNeighbours = @$refLocalNeighbours;
$refRemoteNeighbours = $LocalNeighbours[$i]->{m_RemoteNbr};
@RemoteNeighbours = @$refRemoteNeighbours;
$refRemoteNeighbour = $RemoteNeighbours[$j];
%remoteNeighbour = %$refRemoteNeighbour;
```

The value for the key m_LocalNbr is a pointer to an array, which is a list of hashes, where each hash represents a local neighbor. If there are any remote neighbors, the local neighbor has a key (m_RemoteNbr) whose value points to the reference of an array, which is a list of hashes, each representing a remote neighbor. You will need this data structure if you intend to manipulate it directly. In most cases, however, your task is limited to creating hash lists that define the local and remote neighbors.

The AddLocalNeighbour and AddRemoteNeighbour methods can be used to add neighbors, and the GetLocalNeighbours and GetRemoteNeighbours methods can be used to retrieve information about neighbors.

Local and remote neighbor operation methods

The RIV::Record module provides methods that discovery agents use to perform add and get operations on local and remote neighbors.

The following table identifies and briefly describes the local and remote operation methods that the RIV::Record module provides:

Local and remote neighbor method	Description
AddLocalNeighbour	Adds a local neighbor.
AddLocalNeighbourTag	Adds a tag to a local neighbor.
AddRemoteNeighbour	Adds a remote neighbor.
AddRemoteNeighbourTag	Adds a tag to a remote neighbor.
GetLocalNeighbours	Returns an array of local neighbors.
GetRemoteNeighbours	Returns an array of remote neighbors.
Print	Prints the current record.

See “RIV::Record module reference” on page 231 for the reference (man) pages associated with these methods.

RIV::RecordCache module overview

The RIV::RecordCache module provides an interface to access a record cache file.

RIV::RecordCache constructor

Before accessing the methods that the RIV::RecordCache module provides, you must call the RIV::RecordCache constructor to create and initialize a new record cache file object. The RIV::RecordCache constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the RIV::Agent or RIV::App constructor.

The RIV::RecordCache constructor also takes the name of the record cache file object to be created and an optional path to this object. By default, the optional path is $NCHOME/var/precision

Record cache operation methods

The RIV::RecordCache module provides methods that applications use to perform a variety of operations on records that reside in the cache file.

The following table identifies and briefly describes the record cache operation methods that the RIV::RecordCache module provides:

Record cache method	Description
CacheRecord	Adds a record to the cache file.
GetRecord	Retrieves a record from the cache file.
GetRecords	Retrieves a list of all records residing in the cache file.

See "RIV::RecordCache module reference" on page 236 for the reference (man) pages associated with these methods.

RIV::SnmpAccess module overview

The RIV::SnmpAccess module provides an interface to perform SNMP-related operations on Network Manager MIB trees.

Obtaining SNMP information with the RIV::Agent and RIV::SnmpAccess modules

The following list summarizes how discovery agents should deal with obtaining SNMP information with this version of the Perl API:

- The Helper Server (and ncp_ctrl) must be running so that the Get, GetNext, and GetBulk methods provided by the RIV::Agent and RIV::SnmpAccess modules can make the appropriate queries.
- Discovery agents implemented with this version of the Perl API should use the Get, GetNext, and GetBulk methods provided by the RIV::Agent module to obtain SNMP information from a network device.
- Discovery agents implemented with previous versions of the Perl API and that called the Get, GetNext, and GetBulk methods provided by the RIV::SnmpAccess module will work. There is no need to port these discovery agents to use the Get, GetNext, and GetBulk methods provided by the RIV::Agent module.

RIV::SnmpAccess constructor

Before accessing the methods that the RIV::SnmpAccess module provides, you must call the RIV::SnmpAccess constructor to create and initialize a new RIV::SnmpAccess object. The RIV::SnmpAccess constructor takes a blessed reference to either a discovery agent application object or a client/server application object. These application objects were returned in a previous call to the RIV::Agent or RIV::App constructor.

Maximum number of concurrent asynchronous requests

The RIV::SnmpAccess module provides a MaxAsyncConcurrent variable that sets the maximum number of concurrent asynchronous requests.

Synchronous and asynchronous SNMP operation methods

The RIV::SnmpAccess module provides an interface to the Vertigo SNMP and MIB library functions. Both synchronous and asynchronous variants of each SNMP Get method are provided. The synchronous versions cause the caller to wait until the results are available (or the request has failed). The asynchronous versions all return results via RIV::GetResult. By using this latter method, overlapped I/O may be implemented without the complexity of using Perl threads.

SNMP operation methods

The RIV::SnmpAccess module provides methods that discovery agents and client/server applications use to perform SNMP operations on a network device through the Helper Server. Thus, the Helper Server (and ncp_ctrl) must be running so that the SNMP-related methods can make the appropriate SNMP requests.

The following table identifies and briefly describes the SNMP operation methods that the RIV::SnmpAccess module provides:

SNMP operation method	Description
ASN1ToOid	Converts the specified ASN.1 value to its corresponding OID.
AsyncSnmpGet	Performs an asynchronous SNMP get operation on the specified MIB variable.
AsyncSnmpGetBulk	Performs an asynchronous SNMP get-bulk operation on all MIB objects in the specified MIB table.
AsyncSnmpGetNext	Performs an asynchronous SNMP get-next operation on the specified MIB variable.
GetMibHash	Gets the entire MIB tree by browsing the files that exist in the $NCHOME/mibs directory.
OidToASN1	Converts the specified OID to its corresponding ASN.1 value.
SnmpGet	Performs a synchronous SNMP get operation on the specified MIB variable.
SnmpGetBulk	Performs a synchronous SNMP get-bulk operation on all MIB objects in the specified MIB table.
SnmpGetNext	Performs a synchronous SNMP get-next operation on the specified MIB variable.
SnmpWalk	Performs an SNMP walk operation on a given device, starting at a given MIB variable.
SplitOidAndIndex	Converts the full ASN.1 value into its index and the base OID.

See "RIV::SnmpAccess module reference" on page 240 for the reference (man) pages associated with these methods.

NCP modules overview

The NCP modules provide interfaces that operate on the NCIM topology database and domains.

Summary of Perl API NCP modules

The following table identifies and briefly describes the Perl API NCP modules:

Perl API NCP Module	Description
`NCP::DBI_Factory`	Provides an interface to make it easier to use the standard Perl DBI module to perform operations on the Network Connectivity and Inventory Model (NCIM) topology database.
`NCP::Domain`	Provides an interface to perform operations on NCIM domains.

NCP::DBI_Factory module overview

The `NCP::DBI_Factory` module provides an interface to make it easier to use the standard Perl DBI module to perform operations on the NCIM topology database. Use of this module assumes that you understand the standard Perl DBI module. The `NCP::DBI_Factory` module reads the database login details from DbLogins.*DOMAIN*.cfg, which allows it access to the pre-configured data sources such as NCIM.

DBI handle

The `NCP::DBI_Factory` module provides a method that creates and initializes a new DBI handle. You pass this handle in subsequent calls to the methods that perform operations on the specified NCIM topology database. This DBI handle contains the information needed to connect to the requested NCIM topology database.

Databases that the DBI_Factory module supports

The `NCP::DBI_Factory` module currently supports operations on the following databases:

- Db2®
- Oracle

For all of these databases, table and field names are case-insensitive from the point of view of SQL statements. However, rows returned by both the Db2 and Oracle databases will have all field names in upper case. The `NCP::DBI_Factory` module provides the `toUpper` method that returns a copy of a single row with all lower case field names in upper case.

NCIM Database operation methods

The `NCP::DBI_Factory` module provides methods that client/server applications use to perform a variety of operations on the specified NCIM topology database.

The following table identifies and briefly describes the database operation methods that the `NCP::DBI_Factory` module provides:

NCIM Database method	Description
`createDbHandle`	Creates a standard DBI handle to be used in subsequent calls to the other `NCP::DBI_Factory` methods.
`describeTable`	Returns a sorted array of upper case field names for the specified table or view.

NCIM Database method	Description
extractCmdLineOptions	Allows database login options for the DBI handles to be provided in a common format.
extractHashRefOptions	Extracts database login options from a reference to a hash.
insert_auto_inc_row	Inserts a row into a named table, where the table has an auto incremented column.
insert_row	Inserts a row into a named table.
schema	Returns the schema name associated with the NCIM topology database being used.
setLogHandle	Passes in a log handle associated with an opened file used for logging messages.
setLogLevel	Sets the log level for error and message reporting.
tables	Returns a sorted array of table and view names for the current schema.
timeStamp	Returns the current timestamp in a format suitable for addition to the NCIM topology database.
toUpper	Returns a copy of a hash (a single row retrieved from an NCIM database table) with all field names converted to upper case.

See "NCP::DBI_Factory module reference" on page 253 for the reference (man) pages associated with these methods.

NCP::Domain module overview

The NCP::Domain module provides an interface to perform operations on NCIM Network Manager domains.

NCP::Domain constructor

Before accessing the methods that the NCP::Domain module provides, you must call the NCP::Domain constructor to create a new NCP::Domain object. The NCP::Domain constructor requires the domain name and options for the database connection. The newly created NCP::Domain object encapsulates the attributes associated with the specified domain and database connection.

NCIM domain operation methods

The NCP::Domain module provides methods that client/server applications use to perform a variety of operations on a specified NCIM domains. Some of these operations involve the domainMgr table in the NCIM topology database that resides in the specified domain.

The following table identifies and briefly describes the database operation methods that the NCP::Domain module provides:

NCIM domain database method	Description
clone	Creates a new domain that is a copy of an existing domain.

NCIM domain database method	Description
`create`	Creates an entry in the domainMgr table for this domain if one does not already exist.
`drop`	Removes all references to the specified domain from the domainMgr table.
`id`	Retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain.
`name`	Returns the domain name for the current domain.
`setLogHandle`	Passes in a log handle associated with an opened file used for logging messages.
`setLogLevel`	Sets the log level for error and message reporting.

See "NCP::Domain Reference" on page 276 for the reference (man) pages associated with these methods.

Synchronization with message broker

The Network Manager core components use of message broker is highly multithreaded, whereas Perl applications are single-threaded. Although support for threads was included from Perl 5.005 onwards, this is neither operating system-native, nor POSIX in semantics. Thus, there is no possibility of direct thread-safe integration between the Network Manager and Perl code.

All modules contained in `RIV` (except for the `RIV::Agent` module) assume a single Perl thread. If multiple Perl threads are used, a single thread (the one in which the session was instantiated) must be used for interaction with Network Manager core components. The `RIV::SnmpAccess` module provides both synchronous and asynchronous methods that allow you to perform operations on the MIB tree. The methods `RIV::InputQueueLength` and `RIV::GetResult` are used to query and extract from the application's input queue.

The `RIV::Agent` module provides a multithreading capability into the Perl discovery agent.

See "Using threads in discovery agents" on page 147 for more information.

Installing the Perl API

The Perl API and its associated modules reside in a specific directory.

After installing the Perl API, the required version of Perl and its associated modules reside in the `$NCHOME/precision/perl` directory.

Note: The Perl API is installed when you install the Network Manager core components. Typically, you source the appropriate environment variables in `$NCHOME/precision/env.sh` to set up the required environment to use the Perl API.

Perl builds

Network Manager provides two customized Perl executables as it is necessary to use static linkage against the Network Manager core components libraries.

The first executable is the **ncp_disco_perl_agent** binary. This binary is used by Discovery agents and it is executed automatically by ncp_disco. You would not expect to run this binary directly.

The second executable is the **ncp_perl** binary. You use this binary to develop Perl scripts to customize ITNM or to integrate with other products.

Obtaining SNMP information from a network device

The Perl API allows discovery agents and other client/server applications to obtain SNMP information from a network device through the Helper Server. Typically, this information is retrieved from one or more MIB variables or an entire MIB table.

Helpers retrieve information from the network during a discovery. More specifically, the SNMP helper (ncp_dh_snmp) returns results of an SNMP request such as Get, GetNext, GetBulk, and so forth. The Helper Server can service these SNMP requests directly with cached data or pass on the request to the SNMP helper. The methods that the `RIV::Agent` and `RIV::SnmpAccess` modules provide query the Helper Server. Therefore, the Helper Server must be running so that the methods in these modules can obtain SNMP information from network devices.

Note: It is no longer possible to obtain SNMP information directly from a network device as all queries are now made through the Helper Server.

See the *IBM Tivoli Network Manager IP Edition Administration Guide* for more information on the Helper Server.

In previous versions of the Perl API, the only way to obtain SNMP information from a network device was to call the methods defined in the `RIV::SnmpAccess` module. Futhermore, a discovery agent could obtain SNMP information directly from a network device.

The following list summarizes how discovery agents and other client/server applications should deal with obtaining SNMP information with this version of the Perl API:

- The Helper Server (and ncp_ctrl) must be running so that the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` and `RIV::SnmpAccess` modules can make the appropriate queries.
- Discovery agents implemented with this version of the Perl API should use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module to obtain SNMP information from a network device.
- Discovery agents implemented with previous versions of the Perl API and that called the Get, GetNext, and GetBulk methods provided by the `RIV::SnmpAccess` module will work. There is no need to port these discovery agents to use the Get, GetNext, and GetBulk methods provided by the `RIV::Agent` module.

Perl API modules reference page syntax

The Perl API modules reference pages use a consistent reference page format.

Each Perl API module reference page uses the following format:

- Constructor/Method — This section specifies the name of the constructor or method associated with this Perl API module.
- Synopsis — This section provides the definition for this constructor or method.
- Description — This section provides a description of the functionality that this constructor or method provides.
- Parameters — This section provides descriptions for the input parameters identified in the Synopsis for this constructor or method. If there are no input parameters, this section specifies None.
- Returns — This section provides a description of the value or values that this constructor or method returns. If no values are returned, this section specifies None.
- Notes® — This optional section provides additional information about a constructor or method.
- Example Usage — This section provides an example of how to call this constructor or method.
- See Also — This section provides references to modules or methods that you should be aware of when using this module's constructor or methods.

Chapter 6. Writing discovery agents

Discovery agents retrieve information about devices in the network. They also report on new devices by finding new connections when investigating device connectivity. Discovery agents are used for specialized tasks. For example, the ARP Cache discovery agent populates the Helper Server database with IP address-to-MAC address mappings. You can use the Perl API to write custom discovery agents that perform a variety of useful and specialized tasks, including retrieving information about the connectivity of network entities.

To create custom discovery agents in Perl, you use the `RIV`, `RIV::Agent`, `RIV::Param`, and `RIV::Record` modules.

Before you write a discovery agent

The Perl API is designed to enable the easy creation and prototyping of custom discovery agents. Before writing a custom discovery agent, you need to perform some prerequisite tasks and to be familiar with the discovery process.

Before writing a custom discovery agent, ensure that you have completed these steps:

1. Copied the Agent Definition File (**`.agnt`** file extension) to the `$NCHOME/precision/disco/agents` directory.

   See"Prototype agent definition file template" on page 145 for details about this file.
2. If additional MIBS are required, ensure that they are copied to the `$NCHOME/precision/mibs` directory and that the ncp_mib process is run to import these MIBs into the database.
3. Created new discovery stitchers to process the returns data from the new discovery agent and to add the data into the topology.

   See the *IBM Tivoli Network Manager Reference* for a detailed description of stitchers and the stitcher language.
4. Started the Helper Server (and ncp_ctrl). The Helper Server (and ncp_ctrl) must be running because the SNMP-related methods that the `RIV::Agent` module provides query the Helper Server. It is not possible to obtain SNMP information directly from a network device.

In addition, you should also be familiar with:

- The architecture of the discovery process
- How discovery agents communicate with the DISCO process and the Helper Servers
- The different databases created in the DISCO process to enable discovery agents to work successfully

See *IBM Tivoli Network Manager User Guide* for more information on the previously listed topics.

Writing a discovery agent

Writing a discovery agent requires you to use the `RIV::Agent` and `RIV::Record` modules and to perform a number of prescribed steps. Many discovery agents also use the `RIV` and `RIV::Param` modules.

The following topics describe the steps to follow when writing a discovery agent, using the IP routing discovery agent as an example.

Step 1: Create an agent.pl file

Create an *agent*`.pl` file to contain the Perl code that implements the discovery agent.

Where:

- *agent* — Specifies the name of the file to contain the discovery agent Perl code. For example: ASAgent.pl, TunnelAgent.pl, and NATTextFileAgent.pl are *agent*.pl files that contain the Perl code that implement their respective discovery agents.

Create the *agent*.pl file in the $NCHOME/precision/disco/agents/perlAgents directory.

Step 2: Declare Perl modules

Declare the Perl modules (using use statements) the discovery agent requires. For example:

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent;
```

Step 3: Create a new agent

To begin, you must create a new discovery agent using the RIV::Agent constructor that the RIV::Agent module provides. This constructor sets the name of the discovery agent and also sets up the TCP connections to the DISCO and Helper Server processes. For example:

```
$param = new RIV::Param();
$agent = new RIV::Agent($param, "foo");
```

The example shows that the RIV::Agent constructor consists of two parameters:

- *$param* — Specifies a RIV::Param object. As the example shows, this object is returned in a call to the RIV::Param constructor.
- *$agentName* — Specifies a string that identifies the name of this discovery agent. In this example, the name of the agent is foo.

The RIV::Agent constructor returns a discovery agent application object to the *$agent* variable. You reference all RIV::Agent module methods through this object. For example: $agent->SnmpGet(...), $agent->SnmpGetNext(...), and so forth.

Step 4: Wait for input from the DISCO process

Once the finders detect a network entity that has an OID matching a device that needs to be processed by the discovery agent, the network entity is inserted into the agent's despatch table.

Note: The list of devices supported by the DISCO process is defined in the Agent Definition File.

The DISCO process then sends the record of the device to the agent for processing. This record is received by the Perl discovery agent using the $agent->RIV::GetResult() method. The records received from DISCO are tagged with the string NE. For example:

```
DEVICE: while (1)
{
    my ($tag,$data) = RIV::GetInput(-1);
    if ($tag ne "NE")
    {
        print "Data is not a network entity. Ignoring it!\n";
        next DEVICE;
    }
        my ($tag, $data) = $agent->RIV::GetInput(-1);
        if ($tag ne "NE")
        {
                print "Data is not a network entity. Ignoring it!\n";
                next DEVICE;
        }
        else
        {
                print "This agent is going to process the device!\n";
        }
```

Note: The `RIV::GetInput` function has been deprecated and you should use the `RIV::GetResult` function. The example continues to use the `RIV::GetInput` function to be consistent with the standard scripts which have not yet been updated to use the `RIV::GetResult` function.

Step 5: Create a RIV::Record object

When DISCO sends a record for processing by the discovery agent, the record can be conveniently stored in a data structure. This data structure is referred to as a `RIV::Record` object. You use the `RIV::Record` constructor that the `RIV::Record` module provides to create this object. For example:

```
my $TestNE = new RIV::Record($data);
```

The `RIV::Record` constructor takes the following parameter:

- *$refNE* — Specifies a reference to a hash list.

The record that DISCO sends for processing may be a request from ncp_disco for the agent to terminate. The following code checks for this termination request:

```
my $TestNE = new RIV::Record($data);

    if ($TestNE->{m_TerminateAgent})
    {
        log_msg("Exit Main Loop\n");
        exit(0);
    }
```

The `RIV::Record` constructor returns a `RIV::Record` object.

The `RIV::Record` module provides methods that enable you to easily add and retrieve local and remote neighbors. For example, the following example shows a call to the `AddLocalNeighbour` method.

```
my %localNbr;
$localNbr{'m_IpAddress'} = '1.2.3.4';
$localNbr{'m_IfIndex'} = 2;
$TestNE->AddLocalNeighbour(\%localNbr);
```

The `AddLocalNeighbour` method takes a *$refNbr* parameter that specifies a reference to a hash list. This hash list defines the local neighbor as a set of key value pairs (*varBinds*).

Step 6: Decide if the agent must process the device (pre-mediation layer)

In the pre-mediation layer you must write Perl code to decide if the device needs to be processed by this discovery agent. The Agent Definition File should be used to filter out devices based on the OIDs.

Note: The list of devices supported by the DISCO process is defined in the Agent Definition File.

Typically, this Perl code checks the device's IP address using the `RIV::IsIpValid` method, as shown in the following example:

```
print "Checking if IP address is valid..\n";
if (!RIV::IsIpValid($host))
        {
            print "Device has invalid IP address. Ignoring the record!\n";
            next DEVICE;
        }
```

If the device's IP address is valid, then the discovery agent processes the device. Otherwise, the discovery agent does not process the device. In the example an appropriate message would display to standard output if the device has an invalid IP address.

Step 7: Retrieve device information from the Helper Server (meditation layer)

Next, you must retrieve device information from the Helper Server. This can be achieved using SNMP Get, Telnet, or DNS. For example:

```
{
$refLifindex=$agent->SnmpGetNext($TestNE,'ipAdEntIfIndex');
$refLnetmask=$agent->SnmpGetNext($TestNE,'ipAdEntNetMask');
$refLphysaddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress');
$refRifindex=$agent->SnmpGetNext($TestNE,'ipRouteIfIndex');
$refRtype=$agent->SnmpGetNext($TestNE,'ipRouteType');
$refRnexthop=$agent->SnmpGetNext($TestNE,'ipRouteNextHop');
}
```

The above example uses the `RIV::Agent` method `SNMPGetNext`. The `SNMPGetNext` method takes two parameters:

- *$ne* — Specifies the network entity, which is typically a `RIV::Record` object.
- *$oid* — Specifies a MIB variable, for example, `ipAdEntIfIndex` in the above example.

The above example performs SNMP GET operations on the network entity (NE) in question and retrieves the specified MIB variables. If you prefer, you could substitute the `SnmpGetNext` method with the appropriate methods to allow Telnet or DNS access.

Step 8: Determine local and remote neighbors (processing layer)

In the processing layer, the local and remote neighbors are determined, based on the information from the Mediation layer. A local neighbor is a network interface that resides on the device being discovered. A remote neighbor is something connected to one of these network interfaces.

The following example is taken from the IP routing discovery agent:

```
sub Processing
{
    print "Processing the local neighbours\n";
    foreach $entry (@$refLifindex)
    {
        if (RIV::IsIpValid($entry->{ASN1}))
        {
            my %localNbr;
            $localNbr{'m_IpAddress'} = $entry->{ASN1};
            $localNbr{'m_IfIndex'} = $entry->{VALUE};
            $TestNE->AddLocalNeighbour(\%localNbr);
        }
    }
}
```

Step 9: Sending the processed record to DISCO

Once the network entity has been processed, its record needs to be sent to DISCO. The `RIV::Agent` method `SendNEToDisco` allows you to accomplish this task. The `SendNEToDisco` method takes two parameters:

- *$entity* — Specifies a reference to a hash list that contains the definition of the record to be sent to DISCO. For convenience, the `RIV::Record` module provides an object that serves as a hash list with nested structures for representing local and remote neighbors.
- *$lastRecTag* — Specify the value 1 to indicate that this is the last record for the network entity. Specify the value 0 (zero) to indicate that more records for this network entity are to follow.

  If you use `RIV::Record` objects, the `SendNEToDisco` method ignores this parameter.

Step 10: Running the newly created agent

Before running the newly created agent, make sure that you have:

- Created the agent definition file (*agentName*`.agnt`) in the `$NCHOME/precision/disco/agents` directory. The following example shows the agent definition file for a discovery agent called `CustomPerlAgent`:

  ```
  $NCHOME/precision/disco/agents/CustomPerlAgent.agnt
  ```

- Created or copied the Perl discovery agent script (*agentName*`.pl`) in the `$NCHOME/precision/disco/agents/perlAgents` directory. The following example shows the Perl discovery agent script for a discovery agent called `CustomPerlAgent`:

```
$NCHOME/precision/disco/agents/perlAgents/CustomPerlAgent.pl
```

- Registered the agent with ITNM using the following command:

```
$NCHOME/precision/bin/ncp_agent_registrar -register agentName
```

  Where: *agentName* specifies the name of the discovery agent.

  The following example shows how to register a discovery agent called `CustomPerlAgent`:

```
$NCHOME/precision/bin/ncp_agent_registrar -register
CustomPerlAgent
```

Once the agent has been registered, you should be able to see it and any other registered discovery agents in the Discovery Configuation GUI. Use the Discovery Agent GUI to enable the discovery agent for the next discovery.

For information on the Discovery Configuation GUI, see *IBM Tivoli Network Manager IP Edition Administration Guide*.

Example discovery agents

Typically, a network environment contains multiple discovery agents to support the wide variety of network devices operating in this environment. Thus, the types of discovery agents you can implement with the Perl API is extensive.

The following topics provide simple examples of discovery agents and a skeleton outline of a discovery agent that you can use as a template.

Discovery agent skeleton

The discovery agent skeleton provides an outline of the sections typically implemented in a discovery agent that makes use of the Perl API. This outline also specifies calls to the constructors and methods (for example, `new RIV::Agent`, `RIV::IsIpValid`, `RIV::Agent::SendNEToDisco`, and so forth) typically used in a discovery agent. Use the discovery agent skeleton as a way to start implementing your custom discovery agents.

The following Perl script provides a skeleton outline for a simple discovery agent. Explanations of specific lines follow the skeleton outline:

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent; 1
# Create a new discovery agent
print "Creating a new agent\n"; 2

sub Init{
    my $param=new RIV::Param();
   $agent=new RIV::Agent($param, "PerlDetails"); 3
 # Wait for input from the DISCO process
      print "Entering infinite loop wait for devices for Disco\n"; 4

   DEVICE: while (1){
       my ($tag, $data) = $agent->RIV::GetResult(-1);
       if ($tag ne 'NE'){ 5
          print "Data is not a Network entity Ignoring it!\n";
          next DEVICE;
 }
# Create a RIV::Record object
my $TestNE = new RIV::Record($data);

    if ($TestNE->{m_TerminateAgent})
```

```
    {
        log_msg("Exit Main Loop\n");
        exit(0);
    } 6

# Decide if the agent must process the device (pre-mediation layer)
  ...
  ...
     print "Checking if IP address valid..\n";

     if (!RIV::IsIpValid($host)){ 7
      print "Device has invalid IP address ignoring the record!\n";
     next DEVICE;
 }
 # Retrieve device information from the Helper Server (mediation layer)
  ...
 print "Entering Mediation layer\n"; 8
 Mediation();

 print "Entering Processing layer\n";
 Processing(); 9

 print "Sending Record to Discovery\n";
 $agent->SendNEToDisco($TestNE,0); 10
 }

sub Mediation{ 11
   . . .
   . . .
   # Retrieve the relevant information from the Helper Server using SNMP,
Telnet or DNS.
   . . .
   . . .
 }
 sub Processing{ 12
   . . .
   . . .
   # Determine local and remote neighbors (processing layer) based
   # on the information retrieved in the Mediation Layer.
   # The neighbours are then added to the RIV::Record representing
   # the network entity.
   . . .
   . . .
 }
 Init();
```

The following list explains specific numbered items in the previously listed skeleton outline of a discovery agent:

1. Declare the Perl API modules to use in the discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
2. Calls the `print` operator to display a message to the standard output indicating the creation of a new discovery agent.
3. This is the create or initiate discovery agent section. Creates a new discovery agent with the specified name. The skeleton outline specifies a discovery agent with the name of `PerlDetails`.
4. The discovery agent is ready to receive records from DISCO.
5. Checks that the data records received from DISCO have been tagged with the string NE.
6. Handles a request from ncp_disco to terminate the Perl discovery agent if the test evaluates to true.
7. Checks that the device has a valid IP address.
8. This is the mediation layer section. Gets the relevant SNMP information.
9. This is the processing layer section. Interprets the information to find the local and remote neighbors.
10. Sends the record and filled-out network entity to DISCO.
11. Implements the `Mediation` method.
12. Implements the `Processing` method.

Network entity discovery agent example

The purpose of many discovery agents is to accept network entities from DISCO, process these entities in some way, and then return the result. This example network entity discovery agent provides a skeleton outline for these tasks. Use this example discovery agent skeleton to write discovery agents that need to accomplish these tasks.

The following Perl script provides a skeleton outline for a simple network entity discovery agent:

```
use RIV; 1
use RIV::Param;
use RIV::Agent;
use RIV::Record;

$param = RIV::Param::new();
$agent = RIV::Agent::new($param, "PerlAgent");

while (1)
{
         my ($tag, $data) = $agent->RIV::GetResult(-1);
         next unless ($tag eq "NE"); 2

         foreach my $ne (@{ $data })
         (
             $NE = RIV::Record::new($ne);
             ...
             ...
                $agent->SendToDisco($$ne,1); 3
         )
 )
```

The following list explains specific numbered items in the previously listed discovery agent example:

1. Declare the Perl API modules to use in the discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
2. Checks that the data received from DISCO has been tagged with the string NE.
3. Returns the data to DISCO.

IP routing discovery agent example

The IP routing discovery agent Perl program shows how a simple discovery agent can be written using the Perl API. Study this example to acquire additional knowledge about how to write discovery agents using the Perl API.

The following IP routing discovery agent example uses a representative number of the methods provided in the `RIV::Agent`, `RIV::Record`, and `RIV::Param` Perl API modules. Explanations of specific lines follow the program.

```
use RIV;
use RIV::Param;
use RIV::Record;
use RIV::Agent; 1

print "Creating a new agent\n"; 2
Init();

print "Entering infinite loop wait for devices for DISCO\n"; 3

 DEVICE: while (1){ 4
     my ($tag, $data) = $agent->RIV::GetResult(-1); 5
     if ($tag ne 'NE'){ 6
        print "Data is not a Network entityIgnoring it!\n";
       next DEVICE; 7
     }

     my $TestNE = new RIV::Record($data);
    if ($TestNE->{m_TerminateAgent})
    {
        log_msg("Exit Main Loop\n");
        exit(0); 8
    }
```

```
    $TestNE->{'m_LastRecord'}=1; 9
    $TestNE->{'m_UpdAgent'}="PerlDetails"; 10
    my $host=$TestNE->{'m_IpAddress'}; 11
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. Declare the Perl API modules to use in this discovery agent. The `RIV::Agent` and `RIV::Record` modules are required. The optional `RIV::Param` module is useful for parsing standard and application-specific command line arguments.
2. Calls the `print` operator to display a message to the standard output indicating the creation of a new discovery agent.
3. Calls the `print` operator to display a message to the standard output indicating the program is ready to receive records from the DISCO process.
4. Sets up an infinite loop waiting to receive data from DISCO.
5. Calls the `RIV::GetResult` function to return a data record from DISCO. In this call, the value `-1` is passed signifying that `RIV::GetResult` should "wait forever" to received data records from DISCO before returning.
6. Checks the data record that `RIV::GetResult` returns to the *$tag* variable. If the returned data record has not been tagged with the string NE, then use the `print` operator to display a message to the standard output indicating this data record is not a network entity and thus should not be processed.
7. Continues through the loop to get the next data record from DISCO.
8. Creates a `RIV::Record` object by calling the `RIV::Record` constructor. In this call, the data returned by `RIV::GetResult` to the *$data* variable is passed. This data is actually a reference to a hash list, which is the mechanism used to store network entity records from DISCO.

   The `RIV::Record` constructor returns the newly created `RIV::Record` object to the *$TestNE* variable.

   This code also handles a request from ncp_disco to terminate the Perl discovery agent if the test evaluates to true.
9. Assigns the value `1` to the `m_LastRecord` key in the hash.
10. Assigns the string `PerlDetails` to the `m_UpdAgent` key in the hash.
11. Assigns the value associated with the `m_IpAddress` key in the hash to the `$host` variable.

```
    print "Checking if IP address valid..\n"; 1
    if (!RIV::IsIpValid($host)){ 2
       print "Device has invalid IP address ignoring the record!\n";
       next DEVICE;
    }

    print "Checking if its a router...\n";
    $refNextHop=$agent->SnmpGet($TestNE,'ipForwarding');
    if ($refNextHop->{VALUE} != 1){ 3
       print "Device is not a router!\n";
       next DEVICE;
    }

    print "Entering Mediation layer\n";
    Mediation(); 4

    print "Entering Processing layer\n";
    Processing(); 5

    print "Sending Record to Discovery\n";
    $agent->SendNEToDisco($TestNE,0); 6
} 7

sub Init{
    my $param=new RIV::Param();
    $agent=new RIV::Agent($param, "PerlDetails");
} 8
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code is the Mediation Filter. Check that the device has a valid IP address and also perform an SNMP get for `ipforwarding`. IP routers have a value of `'ipforwarding' =1`.
2. Check that the NE has a valid IP address.
3. Check if the NE is a router. It is a router if the *ipForwarding* value is 1.
4. This section of code is the Mediation Layer. Get the relevant SNMP information.
5. This section of code is the Processing Layer. Interpret the information to find the local and remote neighbors.
6. Send the record to DISCO.
7. The main loop ends.
8. Creates a new agent with the name `PerlDetails`.

```
sub Mediation{ 1

    $refLifindex=$agent->SnmpGetNext($TestNE,'ipAdEntIfIndex'); 2
    $refLnetmask=$agent->SnmpGetNext($TestNE,'ipAdEntNetMask'); 2
    $refLphysaddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress'); 2

    $refRifindex=$agent->SnmpGetNext($TestNE,'ipRouteIfIndex'); 3
    $refRtype=$agent->SnmpGetNext($TestNE,'ipRouteType'); 3
    $refRnexthop=$agent->SnmpGetNext($TestNE,'ipRouteNextHop'); 3

}

sub Processing{ 4
    print "Processing the local neighbours\n";
    for ($j=0;$j<=$#$refLifindex;$j++){
      if (RIV::IsIpValid($refLifindex->[$j]->{ASN1}))
      {
         print $j, "\n";
         my %localNbr
         $localNbr{'m_IpAddress'} = $refLifindex->[$j]->{ASN1}; 5
         $localNbr{'m_IfIndex'} = $refLifindex->[$j]->{VALUE}; 5
         $localNbr{'m_NetMask'} = GetValueForKey($refLnetmask, 5
                     $refLifindex->[$j]->{ASN1});

        $m_1 = $localNbr{'m_IpAddress'};
        $m_2 = $localNbr{'m_NetMask'};
        $localNbr{'m_SubNet'} = inet_ntoa(pack("N", (unpack ("N",
                     inet_aton($m_1)) & unpack("N",
                     inet_aton($m_2))) ));

        for ($i =0; $i <= $#$refLphysaddress; $i++)
        {
          if ($refLphysaddress->[$i]->{ASN1} == $refLifindex
                              ->[$j]->{VALUE})
        {
            $localNbr{'m_LocalNbrPhysAddr'} = $refLphysaddress
                               ->[$i]->{VALUE};
        }
    }
    print $localNbr{'m_IpAddress'}, $localNbr{'m_IfIndex'},
      $localNbr{'m_NetMask'}, "\n";
    $TestNE->AddLocalNeighbour(\%localNbr);
    }
}

print "processing for Remote Neighbours \n"; 6
@localN = $TestNE->GetLocalNeighbours();
for ($j=0;$j<=$#$refRtype;$j++)
{
    if(($refRtype->[$j]->{VALUE} !=2) && RIV::IsIpValid($refRtype
     ->[$j]->{ASN1}))
    {
      $nexthop = GetValueForKey($refRnexthop, $refRtype->[$j]->{ASN1});
      print "next hop = $nexthop for key $refRtype->[$j]->{ASN1}\n";
      if( NotLocalNbr($nexthop))
      {
        my %remoteNbr;
        $remoteNbr{'m_IpAddress'} = $nexthop;
        $Rindex = GetValueForKey($refRifindex,$refRtype->[$j]->{ASN1});
```

```
          AttachLocalNbr(\%remoteNbr, $Rindex);
            }
         }
     }
}
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code implements the Mediation Layer. You should perform all SNMP GET operations in this layer.
2. Get information required for determining local neighbors.
3. Get information required for determining remote neighbors.
4. This section of code implements the Processing Layer. To get the local neighbors, loop through `ipAdEntIfIndex` values. If the ASN1 value is a valid IP address, set the local neighbor tags for local neighbor IP address and `ifIndex`. Set tags for the `netMask` and `physaddress` based on the corresponding values in `ipAdEntIfIndex` and `ifAdEntNetMask`.

   To get the remote neighbors, start looping through the `ipRouteType`. The progam processes only if the route type is not 2 and the ASN1 value is a valid IP address. Then get the corresponding value of the next hop. Make it a remote neighbor if it is not a local neighbor IP address and the remote IP address does not already exist. Use the `ipRouteIfIndex` values to find the local neighbor to attach this remote neighbor to.
5. Add local neighbors to device record `m_IpAddress`, `m_IfIndex`, and `m_NetMask`.
6. This section of code determines and adds remote neighbors.

```
sub GetValueForKey 1
{
    my $refArray = shift;
    my $key = shift;

    for (my $jj=0; $jj<=$#$refArray; $jj++)
    {
      if ($refArray->[$jj]->{ASN1} eq $key)
      {
        return $refArray->[$jj]->{VALUE};
      }
    }
    return 0;
}

sub NotLocalNbr 2
{
    my $ip_in = shift;
    @lN = $TestNE->GetLocalNeighbours();
    foreach $l_nbr (@lN)
    {
      print "RMAA $l_nbr->{m_IpAddress}, $ip_in, \n";
      if ($l_nbr->{'m_IpAddress'} eq $ip_in)
      {
        print "remote neighbour IP same as local neighbour IP \n";
        return 0;
      }
    my @remoteN = $TestNE->GetRemoteNeighbours($l_nbr);
    print @remoteN, "\n";
    foreach my $remoteNbr (@remoteN)
    {
       print "$remoteNbr->{'m_IpAddress'}, $ip_in, \n";
       if ($remoteNbr->{'m_IpAddress'} eq $ip_in)
       {
        print "remote neighbour IP already exists \n";
        return 0;
       }
     }
 }
 return 1;
}
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code returns a value corresponding to the key from an array of `varOps`.
2. This section of code checks if the remote IP is the same as the local neighbor.

```
sub AttachLocalNbr 1
{
    my $refR = shift;
    my $index = shift;
    foreach $lnbr (@localN)
{
       if ($lnbr->{'m_IfIndex'} eq $index)
       {
        print $lnbr->{'m_IfIndex'}, $index, "\n";
        print "$lnbr->{'m_IpAddress'}connected to $refR->{m_IpAddress}\n";
        $TestNE->AddRemoteNeighbour($lnbr, $refR);
       }
  }
}
```

The following list explains specific numbered items in the previously listed IP routing discovery agent Perl program:

1. This section of code finds the appropriate local neighbor to connect the remote neighbor to.

Prototype agent definition file template

A discovery agent requires a discovery agent definition file ($NCHOME/`precision/disco/agents/*.agnt`), regardless of whether the agent is text-based or precompiled. Items that can be defined in this file include when and from where the discovery agent can be run; a list of devices that should be sent to the discovery agent; and the discovery phase at which the discovery agent should complete.

The following is a discovery agent definition file template with required and optional fields. Use this template to create the *.agnt file to be associated with your discovery agent. Explanations of specific lines follow the example.

Note: Where parsing errors occur, the discovery agent definition file rule will be ignored or the default behavior will be assumed.

See the *IBM Tivoli Network Manager IP Edition Administration Guide* for more information on the discovery agent definition file and its associated keywords.

```
--
    -- The following fields are used to initialize the config GUI
    -- and update DiscoAgents.cfg when the agent is first installed
    -- The DiscoAgentDescription keyword provides a way to describe
    -- the discovery agent. The CustomPerlAgent is used as an example.
    DiscoAgentDescription("Agent description goes here.");

    DiscoAgentGUILocked( 0 );
    DiscoAgentClass( 0 );
    DiscoAgentIsIndirect( 0 );
    DiscoAgentPrecedence( 2 );
    DiscoAgentEnabledByDefaultOnPartial( 0 );
    DiscoAgentEnabledByDefault( 0 );
    DiscoAgentDefaultThreads( 1 );

-- Discovery agent type section
DiscoCompiledAgent
{
--
-- Optional "ncp_ctrl" information section
--
-- DiscoExecuteAgentOn("<Machine Name>");
--
-- Devices that should be sent to this agent section
--
DiscoAgentSupportedDevices
(
"Filter Expression"
);
```

```
-- Agent completion phase section
--
DiscoAgentCompletionPhase( n );

-- Mediation filter section
--

DiscoAgentMediationFilter
{
// Optional section containing filters for the mediation layer.
}
}
```

The following list explains specific items in the Agent Definition File template:

- **Discovery agent type section**

  Specifies the agent type. The template specifies the keyword `DiscoCompiledAgent` that denotes a compiled discovery agent. This compiled agent has a corresponding shared library in the `$NCHOME/precision/lib` directory. Possible other values include `DiscoDefinedAgent` and `DiscoCombinedAgent`.

  This section is required.

- **Optional "ncp_ctrl" information section**

  Specifies **`ncp_ctrl`** information. This control information defines when and from what server or computer the discovery agent is to be run. If this line is omitted, the discovery agent will be run on the same server or computer as the DISCO process. Replace *Machine Name* with the name of the server or computer on which the discovery agent is to be run.

  This section is optional.

- **Devices that should be sent to this agent section**

  Specifies the devices that should be sent to the discovery agent. Replace *"Filter Expression"* with valid values that include a range or list of IP addresses to include or exclude, along with a range or list of OIDs to include or exclude. The default is ALL.

  This section is required.

  The following is an example:

```
DiscoAgentSupportedDevices
    (
        "(
           m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.5\.'
         )
         OR
         (
            (
                m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.1\.'
            )
            AND
            (
                m_Description NOT LIKE
                    'IOS \(tm\) C2900XL Software \(C2900XL-C3H2S-M\),
          Version 12.0\(5.3\)WC\(1\), MAINTENANCE INTERIM SOFTWARE'
                AND
                    m_ObjectId <> '1.3.6.1.4.1.9.1.576'
                AND
                    m_ObjectId <> '1.3.6.1.4.1.9.1.577'
                AND
                    m_ObjectId <> '1.3.6.1.4.1.9.1.577'
                AND
                    m_ObjectId <> '1.3.6.1.4.1.9.1.619'
            )
        )"
    );
```

- **Agent completion phase section**

  Specifies at which of the *n* discovery phases should this discovery agent complete.

  This section is required.

The following example shows that the discovery agent should complete at phase 3:

```
--
    -- During which of the n discovery phases should this agent complete?
    --
    DiscoAgentCompletionPhase( 3 );
```

- **Mediation filter section**

  Specifies the mediation layer, which contains, among other items, an optional filter on the mediation layer.

Using threads in discovery agents

Discovery agents written in Perl can experience slow performance because the data retrieved for each device operating in the network is retrieved within a single thread. Thus, the discovery agent spends much of its time idle, waiting for data to be retrieved from a device. The `RIV::Agent` module provides a multithreading capability that improves the performance of discovery agents.

The following topics provide information about the `RIV::Agent` module multithreading capability.

Note: Although Perl itself supports execution of multiple threads, many of the add-on CPAN modules often used with Perl are not thread safe. This means that Perl discovery agents using CPAN modules may need to be restricted to a single thread.

The `RIV::Agent::LockThreads` and `RIV::Agent::UnLockThreads` methods might enable you to use third party Perl modules. These methods allow you to restrict access to a section of a discovery agent to a single thread.

Discovery agent threads example

To overcome the slow performance of discovery agents written in Perl, the `RIV::Agent` module provides a multithreading capability. This capability allows you to serialize execution of specific sections of a discovery agent.

The following example calls the `LockThreads` method to serialize execution of specific sections of a discovery agent:

```
#
    # Begin a serialised section of execution within the Perl agent
    #
    $agent->LockThreads();

    #
    # It is important not to have any fatal errors that could prevent the
    # threads getting unlocked again so wrap the following in an eval block.
    #
    eval
    {
           … only one agent thread executing here at any given time ..
    }
    if ($@)
    {
        warn "Error: $@\nWhen executing serialised block\n";
    }
 #
    # Unlock to allow other threads a chance to execute the above section
    #
    $agent->UnLockThreads();
```

Note: It may be possible to use a non-thread safe Perl module within such a serialized section of the discovery agent. However, no guarantees can be made and results may vary with different modules. So if the results are not successful then the discovery agent may still need to be restricted to a single thread.

Default number of threads

The default number of threads for a Perl discovery agent is defined within its agent definition file.

The number of threads that a Perl discovery agent should use is defined in exactly the same way as that for normal discovery agents, by modifying the `m_NumThreads` attribute in the `DiscoAgents.cfg` configuration file. The default number of threads for a Perl discovery agent is specified in the agent's definition file as follows:

```
DiscoAgentDefaultThreads( 10 );
```

Note: In order for a Perl discovery agent to use multiple threads, a `DiscoAgentDefaultThreads` entry must be defined in its agent definition file. If such an entry does not exist, then the value of the `m_NumThreads` attribute in the `DiscoAgents.cfg` configuration file will be ignored.

Chapter 7. Accessing component databases

Network Manager provides component databases that store specific categories of information. Each component schema can consist of one or more databases and each database one or more tables. For example, the ncp_class database consists of one database and three tables. You use the `RIV::OQL` Perl API module to access or modify records in these component databases.

To access or modify records in the Network Manager component databases, you use the `RIV::OQL` Perl API module. Typically, you will also make use of the `RIV::App` and `RIV::Param` Perl API modules.

See the *IBM Tivoli Network Manager Reference* for information about the component databases you can access with the `RIV::OQL` module.

Object Query Language

Object Query Language (OQL) is a version of the Structured Query Language (SQL) that has been designed for use in Network Manager. The components create and interact with their databases using OQL.

Use OQL to create new databases or insert data into existing databases (to configure the operation of Network Manager components) by amending the component schema files. You can also issue OQL statements using the OQL Service Provider, for example, to create or modify databases, insert data into databases and retrieve data.

For more information about the OQL schema used by Network Manager, see the *IBM Tivoli Network Manager Reference*.

The OQL Service Provider is described in the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Differences between OQL and Structured Query Language

Network Manager uses OQL to transfer data between and communicate with its internal databases. OQL is an object-based version of Structured Query Language (SQL) that was designed specifically around the operational needs of the Network Manager architecture.

The following items identify the main differences between OQL and SQL:

- OQL has the ability to support object referencing within database tables. Thus, it is possible to have objects nested within objects.
- Not all SQL keywords are supported within OQL. Thus, irrelevant keywords have been removed for the OQL syntax.

Note: Before using the Perl API to access the component databases, make sure you understand the available component databases and the Object Query Language. See *IBM Tivoli Network Manager Reference* for details related to the component databases. See *IBM Tivoli Network Manager Reference* for details related to the Object Query Language.

Actions that can be performed on component databases

The Perl API provides utilities that allow you to query any of the Network Manager component databases and to retrieve and control the information stored in them.

The `RIV::OQL` Perl API module allows you to perform operations on the component databases, including:

- Inserting new entries into the component databases
- Modifying existing device attributes
- Deleting entries from the component databases

To access a particular component database, the service in which the database resides must be running. For example, if you want to access a component database that resides in DISCO, **ncp_disco** must be running. Likewise, to access a CLASS database, **ncp_class** must be running. For a full listing of services to which you can connect, see "RIV::OQL Constructor" on page 214.

After you create a `RIV::OQL` object based on a selected service, you can perform one of four actions on a component database:

- SELECT
- INSERT
- UPDATE
- DELETE

A SELECT statement returns records, while the other statements do not return any records.

The `RIV::OQL` module allows you to:

- Access retrieved records using the `RIV::GetResult` method.
- Print these retrieved records using the `RIV::OQL::Print` method.
- Create new databases and tables with the `RIV::OQL::CreateDB` and `RIV::OQL::CreateTable` methods.

In addition to the previously described convenience methods, you can use the `RIV::OQL::Send($statement, $returnResults)` method, where `$statement` is any valid OQL statement, and `$returnResults` equals:

- 1 — For queries that return results. For example, select and show.
- 0 — For queries that do not return results. For example, insert.

Example Perl scripts that operate on component databases

Use the following examples as guides to writing Perl scripts that access the Network Manager component databases.

The oql_example.pl example script

The **oql_example.pl** script demonstrates the use of Perl to parse the `/etc/hosts` file and input any devices listed therein into the **finders.despatch** database, providing the IP address listed is valid.

The **oql_example.pl** script uses some of the methods provided in the `RIV::Param` and `RIV::OQL` Perl API modules.

```
#!/opt/netcool/precision/Solaris2/bin/ncp_perl
 use strict;
 use RIV;
 use RIV::Param;
 use RIV::App;
 use RIV::OQL; 1

 my $param = new RIV::Param() or 2
 die "RIV::Param::new failed";

 my $app = new RIV::App($param, "ncp_test:oql") or 3
 die "Can't create RIV application session";

 my $oql = new RIV::OQL($app, "Disco") or 4
 die "Can't create RIV OQL session";


 open (INPUT, "/etc/hosts") 5
 or die "Could not open /etc/hosts: $!\n":
 my $number_records =0; 6
 my $finder = "PerlFileFinder";
 my $sep = "'";
 my $stat;
 while (<INPUT>){
```

```
     next if (/^#/ or /^--/); 7
     my ($ipadd, $ipname)= /^(\S+)\s+(\S+)/; 8

     if (RIV::IsIpValid($ipadd)){ 9
        my %record = (m_Creator => $sep.$finder.$sep, m_Name => 10
       $sep.$ipname.$sep, m_IpAddress => $sep.$ipadd.$sep, );
     $oql->Insert('finders', 'despatch', \%record); 11
     $number_records ++;
     }
 }
 print STDOUT "Number of Records input = ", $number_records; 12
```

The following list explains specific numbered items in the previously listed Perl script example:

1. Declare the Perl API modules to use in the **oql_example.pl** script. This script makes use of the RIV, RIV::Param, RIV::App, and RIV::OQL modules. The **oql_example.pl** script also makes use of the `use strict` pragma. The `use strict` pragma enforces good programming practices, including enforcing the declarations of any new variables with `my`.
2. Creates and initializes a new `Param` object. If the `Param` object cannot be created the script stops. In this call to the `RIV::Param` constructor, no arguments are specified. This means that there are no application-specific command line arguments. However, the standard command line arguments that the `RIV::Param` module provides are available once the `Param` object is created.
3. Creates a new client/server application object using the `RIV::App` constructor. This call to the `RIV::App` constructor takes two parameters:
   - `RIV::Param` — Specifies a `RIV::Param` object reference. This object was returned to `$param` by the `RIV::Param` constructor.
   - `$progname` — Specifies a string that uniquely identifies this application. By convention, the application name should start with `ncp_`. In the example, the specified application name is `ncp_test:oql`.

     If the client/server application object cannot be created, the script stops.
4. Creates a new OQL object on the service type Disco. If the OQL object fails to create, the script stops.
5. Opens and reads the file `/etc/hosts` using the name INPUT as a file handle.
6. Initializes variables.
7. Ignores lines beginning with # or -- characters.
8. Browse each line and get the IP address and name, filtering out any spaces in between.
9. Checks the validity of the IP address.
10. If the IP address is valid, the value pairs `m_Creator`, `m_Name`, and `m_IpAddress` are put in the *%record* hash.
11. Inserts the record in the despatch table of the finders database.
12. After the loop completes, prints to standard output the number of records input into the finders.despatch table. The finders database is defined in the DiscoSchema.cfg file.

    See the *IBM Tivoli Network Manager IP Edition Administration Guide* for information about the finders database.

OQL example script

The OQL example script shows how to create an OQL session and perform several operations on the MODEL database.

```
# $PRECISION_HOME/bin/ncp_perl
 use RIV;
 use RIV::Param;
 use RIV::App;
 use RIV::OQL; 1

 my $param = new RIV::Param() 2
 or die "RIV::Param::new failed";

 my $app = new RIV::App($param, "ncp_test:oql"); 3
```

```
 or die "Can't create RIV application session";

 my $oql = new RIV::OQL($app, "Model"); 4
 or die "Can't create RIV OQL session";

 my $stat ='insert into master.entityByName (EntityName, Description, 5
ClassName) values ("bar", "This is a switch", "Switch");';
 $oql->Send($stat, 0);

 $stat = 'select * from master.entityByName;'; 6
 $oql->Send($stat);
 my ($type, $data) = $oql->RIV::GetResult(10); 7
 $oql->Print($data);

 $oql->Select('master', 'entityByName', 'ALL'); 8
 ($type, $data) = $oql->RIV::GetResult(10);
 $oql->Print($data);


 $oql->Delete(master, entityByName, "ClassName = 'Switch'"); 9

 my %insert_rec; 10
 $insert_rec{EntityName} = "'foo'";
 $insert_rec{Description} = "'This is a router'";

 $oql->Insert('master', 'entityByName', \%insert_rec);

 $oql->Select('master', 'entityByName', 'ALL'); 11
 ($type, $data) = $oql->RIV::GetResult(10);
 $oql->Print($data);

 $oql->CreateDB("PerlDB"); 12

 my %table_columns = (m_IpAddress=> "text", m_Name=> "text"); 13
 $oql->CreateTable("PerlDB", "PerlTable", \%table_columns);
 my %dummy_entry = ("m_IpAddress"=> "'8.9.10.11'", "m_Name" => "'dummy'"); 14
 $oql->Insert('PerlDB', 'PerlTable', \%dummy_entry);

 $oql->Select('PerlDB', 'PerlTable', 'm_Name'); 15
 ($type, $data) = $oql->RIV::GetResult(10);
 $oql->Print($data);
```

The following list explains specific numbered items in the previously listed Perl script example:

1. Declare the Perl API modules to use in the OQL example script. This script makes use of the `RIV`, `RIV::Param`, `RIV::App`, and `RIV::OQL` modules.
2. Reads and parses the command line. The standard arguments are hidden.
3. Creates a new `RIV` application object. If the creation of this object fails, the script stops.
4. Creates an `OQL` object with service `Model`. If the creation of this object fails, the script stops.
5. Inserts a record into MODEL using the `Send` method.
6. Determines what records there are in the MODEL database using the `Send` method.
7. Gets and prints the records.
8. Determines what records there are in the MODEL database using the `Select` method.
9. Deletes the record using the `Delete` method.
10. Inserts a new record.
11. Checks if the records were deleted or inserted.
12. Creates a database called `PerlDB`.
13. Creates a table in the `PerlDB` with columns `m_IpAddress` and `m_Name`.
14. Inserts a record into the `PerlDB` database.
15. Selects the entries in the user defined database to verify that the database table has been created and the record inserted.

Chapter 8. Performing SNMP queries

The RIV::SnmpAccess module allows client/server scripts that use the Perl API to retrieve SNMP information from a network device through the SNMP helper.

To write client/server Perl scripts that retrieve SNMP information from network devices you typically use the RIV::Param, RIV::App, and RIV::SnmpAccess modules.

Note: Discovery agent Perl scripts should not use the RIV::SnmpAccess module. Discovery agent Perl scripts use the RIV::Agent module, which provides its own methods to perform SNMP operations through the SNMP helper.

Using get methods to obtain SNMP information from a device

The Perl API, specifically the RIV::SnmpAccess module, provides a number of get methods for retrieving SNMP information from a particular device. You can make these get SNMP information requests either synchronously or asynchronously because the RIV::SnmpAccess module provides both synchronous and asynchronous versions of these get methods.

The following table summarizes which methods to call in a client/server Perl script.

Synchronous/ asynchronous method	Description	Level of SNMP information accessed
SnmpGet and AsyncSnmpGet	The caller specifies a valid IP address for the particular device and the MIB variable of interest. These methods return the specified MIB variable for the specified device.	These methods retrieve a single MIB variable. If you pass a MIB table (instead of a single MIB variable) to these methods, only the first entry in this MIB table is returned.
SnmpGetNext and AsyncSnmpGetNext	The caller specifies a valid IP address for the particular device and a MIB table of interest. These methods return the specified MIB table for the specified device.	These methods retrieve an entire MIB table that contains multiple variables (for example, ifTable).
SnmpGetBulk and AsyncSnmpGetBulk	The caller specifies a valid IP address for the particular device and the MIB variables of interest. These methods return the specified MIB variables for the specified device.	These methods retrieve multiple MIB variables (for example, ifDescr, ifType, and ifSpeed).

Making synchronous and asynchronous SNMP get requests

The Perl API provides two ways to make SNMP get requests: synchronous and asynchronous. You make these SNMP get requests by calling the get request methods that the RIV::SnmpAccess module provides.

The following list briefly describes the two ways to perform SNMP get requests:

- Synchronous — Each successive transmission of data requires a response to the previous transmission before a new one can be initiated.
- Asynchronous — Each transmission of data proceeds independently until one transmission needs to interrupt another one with a request.

When writing a client/server Perl script that makes a synchronous SNMP GET request, the request is made and the caller will not be able to perform other tasks until the specified MIB variable has been retrieved. The information that is returned will have an attached tag so that you know what it is referring to. The tag will be whatever you have specified it to be in the synchronous get method.

Unlike a synchronous SNMP GET request, an asynchronous SNMP GET request is multithreaded. Thus, you are free to perform other tasks while waiting for a response when using an asynchronous SNMP GET request. In the Perl API, the caller can specify the number of threads. When retrieving SNMP information from a large device, use 10 threads.

Example SNMP GET access script

The SNMP GET access example Perl script shows how to retrieve SNMP information using several of the SNMP get methods — both synchronous and asynchronous — that the `RIV::SnmpAccess` module provides. Use this example as a model for writing your own client/server Perl scripts that retrieve SNMP information for specific devices from a single MIB variable, multiple MIB variables, and a MIB table.

Declare Perl API modules and variables

This section of the SNMP GET access example Perl script declares the Perl API modules to be used as well as a number of variables. Use this part of the example script as a guide to setting up client/server Perl scripts that will retrieve SNMP information.

The SNMP GET access example Perl script declares the Perl API modules to be used and a number of variables as follows:

```
#!$NCHOME/bin/ncp_perl
 use strict; 1
 use RIV;
 use RIV::Param;
 use RIV::App;
 use RIV::SnmpAccess; # qw (RivSnmpResultOk); 2

 $RIV::SnmpAccess::MaxAsyncConcurrent = 40; 3

 my $Ttype = "TEST"; 4
 my $Verbose;
 my @_Usage = ("node" [async]);
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Declares the `strict` pragma with the `use` directive. The `strict` pragma enforces good programming practices, including enforcing the declarations of any new variables with `my`.
2. Specify the `use` directive to declare the Perl API modules to be used. In this case, use the `RIV`, `RIV::Param`, `RIV::App`, and `RIV::SnmpAccess` modules.
3. Sets the *MaxAsyncConcurrent* `RIV::SnmpAccess` module variable to the value 40. This module variable sets the maximum number of concurrent asynchronous SNMP get requests.
4. Declare the following `my` variables:
   - *$Ttype* — Stores a string that identifies whether the SNMP GET access is synchronous or asynchronous. Later sections of the SNMP GET access example script use this variable in calls to the `print` operator. The variable gets set initially to the string TEST.
   - *$Verbose* — Specifies how the script progress details are to be displayed. The `-v` option displays verbose progress details. This variable is defined with the `RIV::Param` module, specifically with the `Usage` method.
   - *@_Usage* — Specifies the usage string suffixes `node` and `async`.

Create and initialize a RIV::Param object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::Param` object. Use this part of the example script as a guide to creating and initializing new `RIV::Param` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::Param` object as follows:

```
my $param = new RIV::Param({"-v"=> [ $RIV::Param::NoArg, \$Verbose ],}, 1
        \@_Usage) or
die "RIV::Param::new failed"; 2

my $node = shift @ARGV;
my $what = shift @ARGV;
$what = "" unless defined $what;

$param->Usage(1) unless (defined $node && $node ne "");
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Creates and initializes a new `RIV::Param` object by calling the `RIV::Param` constructor.

   Note: The standard arguments (for example, `-domain`, `-debug`, and so forth) are hidden.
2. If the constructor fails to create and initialize the new `RIV::Param` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::Param` constructor failed) to the standard error stream.

Create and initialize a RIV::App object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::App` object. Use this part of the example script as a guide to creating and initializing new `RIV::App` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::App` object as follows:

```
my $app = new RIV::App($param, "ncp_test:snmp") or 1

die "Can't create RIV application session" unless (defined $app); 2
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Creates and initializes a new `RIV::App` object by calling the `RIV::App` constructor. This call to the `RIV::App` constructor takes the following parameters:
   - `RIV::Param` — Specifies a reference to a `RIV::Param` object. In this example, the newly created `RIV::Param` object is returned to the `my $param` variable in a previous call to the `RIV::Param` constructor.
   - `$progname` — Specifies a string that uniquely identifies an application. In this example, the application name is identified by the string `ncp_test:snmp`.
2. If the constructor fails to create and initialize the new `RIV::App` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::App` constructor failed) to the standard error stream.

Create and initialize a RIV::SnmpAccess object

This section of the SNMP GET access example Perl script creates and initializes a new `RIV::SnmpAccess` object. Use this part of the example script as a guide to creating and initializing new `RIV::SnmpAccess` objects in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script creates and initializes a new `RIV::SnmpAccess` object as follows:

```
my $snmp = new RIV::SnmpAccess($app) or 1
die "Can't create RIV SNMP session"; 2
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Creates and initializes a new `RIV::SnmpAccess` object by calling the `RIV::SnmpAccess` constructor. Upon successful completion, the `RIV::SnmpAccess` constructor returns a `RIV::SnmpAccess` object to the `my $snmp` variable.

   This call to the `RIV::SnmpAccess` constructor takes the following parameter:

   - `$rivSession` — Specifies a reference to `RIV::App` object returned in a previous call to the `RIV::App` constructor. In this example, the newly created `RIV::App` object is returned to the `my $app` variable in a previous call to the `RIV::App` constructor.
2. If the constructor fails to create and initialize the new `RIV::SnmpAccess` object, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the `RIV::Snmp` constructor failed) to the standard error stream.

Check the device IP address and node name

This section of the SNMP GET access example Perl script checks for the device's IP address and node name. Use this part of the example script as a guide to writing code that checks for a device's IP address and node name in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script checks for a device's IP address and node name as follows:

```
my $nodeIP = $node; 1
if ($node !~ /^\d+\.\d+\.\d+\.\d+$/) { 2
$nodeIP = gethostbyname($node); 3
$nodeIP = inet_ntoa($nodeIP) if (defined $nodeIP) or
die "Can't find IP address for '$node'"; 4
}
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Assigns the value stored in the `my $node` variable to the `my $nodeIP` variable. The `my $node` variable was set after the call to the `RIV::Param` constructor.

   See “RIV::Param Constructor” on page 225 for more information.
2. Determines if an IP address or node (host) name was specified.
3. Gets the IP address from the node name by calling the `gethostbyname` and `inet_ntoa` functions.
4. If the `defined` function verifies that the value in `$nodeIP` is `undef`, consider this a fatal error and call the `die` function. The `die` function prints out an appropriate message (in this case, that the IP address for this device cannot be found) to the standard error stream.

Determine which SNMP GET requests to run

This section of the SNMP GET access example Perl script determines which SNMP GET requests — synchronous or asynchronous — to run. Use this part of the example script as a guide to writing code that sets up an appropriate condition to run either the synchronous or asynchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up a condition to run either the synchronous or asynchronous SNMP GET requests as follows:

```
if ($what =~ /async/) { 1

$Ttype = "ASYNCTEST";

AsyncTests();

exit 0;

}

SyncTests(); 2

exit 0;
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Checks the input parameter to determine if the asynchronous tests (using the asynchronous SNMP GET requests) should be run.
2. By default, the synchronous tests (using the synchronous SNMP GET requests) should be run.

Perform asynchronous SNMP GET requests

This section of the SNMP GET access example Perl script sets up the logic to run the asynchronous SNMP GET requests. Use this part of the example script as a guide to writing code that makes use of some of the asynchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic and runs the asynchronous SNMP GET requests as follows:

```
sub AsyncTests {
    my $sTag = "GETNEXT_$node"; 1
    print "$Ttype: call AsyncSnmpGetNext($sTag, $nodeIP, \"\", \"ifDescr\")\n";


    AsyncSnmpGetNext($sTag, $nodeIP, "", "ifDescr") or
    die "AsyncSnmpGetNext() failed";

    my ($tag, $data) = $snmp->GetInput(-1) or 2
    die "Unexpected tag ’$tag’";


    foreach my $varop (@{ $data->[0] })
    {
        PrintVarOp($varop);
    }


    $sTag = "GET_$node"; 3

    print "$Ttype: call AsyncSnmpGet($sTag, $nodeIP, \"\", \"ifDescr\", \"2\")\n"

    AsyncSnmpGet($sTag, $nodeIP, "", "ifDescr", "2")

    or die "AsyncSnmpGet() failed" ;

    ($tag, $data) = $app->GetInput(-1)
    or die "Unexpected tag '$tag'" unless ($tag eq "SNMP_$sTag");
    PrintVarOp($data->[0]);
}
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Performs an SNMP GETNEXT asynchronous request (by calling the `AsynchSnmpGetNext` method).
2. Receives the results, using the `$snmp->RIV::GetResult()` method, and then prints these results.

   Note: The `RIV::GetInput` function has been deprecated and you should use the `RIV::GetResult` function. The example continues to use the `RIV::GetInput` function to be consistent with the standard scripts which have not yet been updated to use the `RIV::GetResult` function.
3. Performs an SNMP GET asynchronous request (by calling the `AsynchSnmpGet` method). Receives the results using the `$snmp->RIV::GetResult()` method. Then checks the tag and prints the results.

   Note: The `RIV::GetInput` function has been deprecated and you should use the `RIV::GetResult` function. The example continues to use the `RIV::GetInput` function to be consistent with the standard scripts which have not yet been updated to use the `RIV::GetResult` function.

Perform synchronous SNMP GET requests

This section of the SNMP GET access example Perl script sets up the logic to run the synchronous SNMP GET requests. Use this part of the example script as a guide to writing code that makes use of some of the synchronous SNMP GET requests in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic and runs the synchronous SNMP GET requests as follows:

```
sub SyncTests {

print "$Ttype: call SnmpGetNext($nodeIP, NULL, ifDescr)\n"; 1
my ($vap) = $snmp->SnmpGetNext($nodeIP, "", "ifDescr") or
die "SnmpGetNext on ifDescr table for '$node' failed";

foreach my $varop (@{ $vap })
{
    PrintVarOp($varop);
}

print "$Ttype: call SnmpGet($nodeIP, NULL, ifDescr,1)\n"; 2
$vap = $snmp->SnmpGet($nodeIP, "", "ifDescr",1) or
die "SnmpGetNext on ifDescr table for '$node' failed";

PrintVarOp($vap);

print "$Ttype: call SnmpGetBulk($nodeIP, NULL,\@oids,8,100)\n";
my @oids=('sysDescr', 'sysContact', 'sysUpTime', 'ipInReceives',
'ipOutRequests', 'ipOutDiscards', 'ipForwDatagrams',
'tcpCurrEstab', 'ifDescr');

($vap) = $snmp->SnmpGetBulk($nodeIP, "", \@oids, 8, 100); 3
foreach my $varop (@{ $vap })
{
    PrintVarOp($varop);
}
}
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Performs an SNMP GETNEXT synchronous request (by calling the `SnmpGetNext` method).
2. Performs an SNMP GET synchronous request (by calling the `SnmpGet` method).
3. Performs an SNMP GETBULK synchronous request (by calling the `SnmpGetBulk` method).

Print the SNMP varops

This section of the SNMP GET access example Perl script sets up the logic to print the SNMP varops. Use this part of the example script as a guide to writing code that prints the SNMP varops in client/server Perl scripts that retrieve SNMP information.

The SNMP GET access example Perl script sets up the logic to print SNMP varops as follows:

```
sub PrintVarOp { 1
    my ($vp) = @_;

   my $asn1 = $vp->{ASN1};
        my $value = $vp->{VALUE};
        my ($oid, $index, $name) = $snmp-> SplitOidAndIndex ($asn1);
        print "$Ttype: $name.$index = $value\n";
}
```

The following list explains specific numbered items in the previously listed section of the SNMP GET access Perl script example:

1. Prints the SNMP varops.

Chapter 9. Writing and integrating Perl applications with third-party products

The Perl API allows you to write Perl applications (for example, Listeners) that you can then integrate with third-party products.

Listener applications

A Listener is an application written for a specific Network Manager database. The purpose of a Listener is to "listen" and respond to record events that occur in the associated database.

Record events in the database include updates to existing records, additions of new records, and deletions of existing records. A Listener application can process these record events in order to:

- Update an external database
- Send email to an appropriate administrator or end user based on the event type
- Integrate with a variety of third-party products or applications

Users of the Perl API can also make use of the mail modules (for example, `Mail::Mailer`) to email database record events. Listener applications, through the `RIV::OQL` module, can send a stream of data into HTML, CGI scripts, and XML data.

Note: Communication with external databases — such as, Oracle® or Sybase® — can be done using the Perl DBI module.

In the record received from the Listener there is a tag for `Action Type` that defines the action performed. For example, a record returned with an action type of 2 indicates that the listener had picked up a record deletion. The actions are summarized in the table below.

Table 74. Listener actions

Tag	Action
0	Insert
1	Update
2	Delete

Note: The listener must be associated with a subject. For example, to listen to events the subject must be `ITNM/EVENT/NOTIFY`.

Example Listener script

The Listener example script shows how to "listen" to record insertions, deletions, and updates in the MODEL topology database. Use this example as a model for writing your own Listener applications using the Perl API.

Declare Perl API modules and variables for Listener

This section of the Listener example script declares the Perl API modules to be used. Use this part of the example script as a guide to declaring the Perl modules used with Listener applications.

The Listener example script declares the Perl API modules to be used as follows:

```
use RIV;
```

```
use RIV::Param;
use RIV::App; 1
```

The following list explains specific numbered items in the previously listed section of the Listener Perl script example:

1. Declare the Perl API modules to be used with the `use` directive, specifically, `RIV`, `RIV::Param`, and `RIV::App`.

Create and initialize a RIV::Param object for Listener

This section of the Listener example script creates and initializes a new `RIV::Param` object. Use this part of the example script as a guide to creating and initializing new `RIV::Param` objects in Listener applications.

The Listener example script creates and initializes a new `RIV::Param` object as follows:

```
$param = RIV::Param::new(); 1
```

The following list explains the specific numbered item in the previously listed section of the Listener Perl script example:

1. Creates and initializes a new `RIV::Param` object by calling the `RIV::Param` constructor. Upon successful completion, the `RIV::Param` constructor returns a `RIV::Param` object to the `$param` variable. This `RIV::Param` object is then passed as a parameter to the `RIV::App` constructor.

Create and initialize a RIV::App object for Listener

This section of the Listener example script creates and initializes a new `RIV::App` object. Use this part of the example script as a guide to creating and initializing new `RIV::App` objects in Listener Perl scripts.

The Listener example script creates and initializes a new `RIV::App` object as follows:

```
$app = RIV::App::new($param, "model_listener"); 1
```

The following list explains the specific numbered item in the previously listed section of the Listener Perl script example:

1. Creates and initializes a new `RIV::App` object by calling the `RIV::App` constructor. This call to the `RIV::App` constructor takes the following parameters:
   - `RIV::Param` — Specifies a reference to a `RIV::Param` object. In this example, the newly created `RIV::Param` object is returned to the `$param` variable in a previous call to the `RIV::Param` constructor.
   - `$progname` — Specifies a string that uniquely identifies an application. In this example, the application name is identified by the string `model_listener`.

Bind the RIV::App object to the message broker subject for Listener

Network Manager uses the message broker publish and subscribe messaging system to enable processes to communicate with each other. This section of the Listener example script binds the newly created `RIV::App` object to message broker. Use this part of the example script as a guide to binding `RIV::App` objects to message broker.

The Listener example script binds the newly created `RIV::App` object to message broker as follows.

See the *IBM Tivoli Network Manager IP Edition Administration Guide* for more information on message broker.

```
$ok = $app->RIV::AddSubject('ITNM/MODEL/NOTIFY','model'); 1
print $ok, "\n"; 2
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

1. Calls the AddSubject virtual method to bind the RIV::App object to message broker. The AddSubject virtual method takes two parameters:
   - $subject — Specifies the message broker subject to which the RIV::App object binds. In this call, the message broker subject is ITNM/MODEL/NOTIFY.

     Note: If you wanted the Listener application to listen to events, then $subject would be ITNM/EVENT/NOTIFY.
   - $tag — Specifies the tag to be appended to USER_, which describes the message returned through the RIV::GetResult method. In this call, the tag is model.

   Upon successful completion, the AddSubject virtual method returns the value 1.
2. Calls the print operator to print the value that the AddSubject virtual method returns to the standard output.

Write database records to a log file

This section of the Listener example script sets up an appropriate loop for "listening" to records inserted, deleted, or updated in the MODEL database and then sending the information to a log file. Use this part of the example script as a guide to setting up appropriate loops to capture record activity and then send this activity to some log file in Listener Perl scripts.

The Listener example script sets up an appropriate loop for capturing record activity in the MODEL database as follows:

```
open(LOGFILE, ">>model.log)"; 1
while(1){ 2
    my ($tag, $data) = $snmp->GetResult(-1);
    foreach $key (@$data){
    foreach $rec (keys %$key){
      {
        print LOGFILE "$rec : $key->{$rec}","\n"; 3
      }
    }
}
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

1. Calls the open operator to open the file handle LOGFILE for output (or appending) to the new (or existing) file model.log.
2. Sets up a while loop that executes until all inserted, deleted, and updated database records are processed and written to the model.log file.
3. Writes the desired information to the model.log file. However, if you want to send the information to a third-party database management application, such as Sybase or Oracle, or a trouble-ticketing system, such as ClearQuest®, use the Perl DBI module. To accomplish this, replace this line of code with the DBI connect method and then send the information.

Send database records to different applications

This section of the Listener example script sets up an appropriate loop for "listening" to records inserted, deleted, or updated in the MODEL database and then sending the information to different applications using the Perl DBI module. Use this part of the example script as a guide to setting up appropriate loops to capture record activity and then sending that information to different applications using the Perl DBI module.

The Listener example script sets up an appropriate loop for capturing record activity in the MODEL database and then sending that information to an Oracle database as follows:

```
 use DBI; 1
.
.
.
     $dbname = 'modelEntities'; $user = 'foo'; 2
     $password = 'foobar'; $dbd = 'Oracle';
     $dbh = DBI->connect($dbname, $user, $password, $dbd);
     . . .
     . . .
     $dbh->do($statement);
```

The following list explains the specific numbered items in the previously listed section of the Listener Perl script example:

1. Declares use of the Perl DBI module. See the Perl DBI documentation for more information.
2. Sets up the appropriate code to send database information to the Oracle database. Note the use of the DBI `connect` method.

Chapter 10. RIV Modules Reference

Each RIV module provides constructors and methods used in the Perl scripts that you implement to create custom discovery agent and other client/server applications.

To implement Perl scripts using the RIV modules, you must be familiar with the constructors and methods that each module provides. These constructors and methods are described in manual (reference) page format.

RIV module reference

The RIV module is a container for a set of modules that support implementation of Perl applications on IBM Tivoli Network Manager IP Edition.

The RIV module provides variables, functions, and virtual methods that the Perl API application modules — `RIV::Agent` and `RIV::App` — use.

RIV module synopsis

The RIV module synopsis provides summary calls to the variable, functions, and virtual methods that the `RIV::Agent` module and `RIV::App` module can use.

Synopsis

```
# Add an IO handle
$ok = $rivSession->AddIoHandle($fileHandle, $tag);

# Load the RIV module.
use RIV;

# Call the RIV::RivDebug function.
$RIV::DebugLevel;
RIV::RivDebug($lvl, $debugString);

# Call the RIV::RivMessage function.
$RIV::MessageLevel;
RIV::RivMessage($msglvl, $messageString);

# Call the RIV::RivError function.
RIV::RivError($class, @errorMessageStrings);

# Call the RIV::InputQueueLength function.
$qlen = RIV::InputQueueLength();

# Call the RIV::GetResult function. Note the optional $waitTime
# parameter. Note also that $rivSession stores the application object
# returned in a previous call to the RIV::Agent or RIV::App constructor.
($tag, $value, $domain) = RIV::GetResult($waitTime);
($tag, $value, $domain) = $rivSession->RIV::GetResult([ $waitTime ]);

# Call the RIV::GetResultSet function. Note the optional $waitTime parameter.
$rsKey = $rivSession->RIV::GetResultSet([ $waitTime ]);

# Call the RIV::InputFilter function.
$ok = RIV::InputFilter($pattern, $function);

# Call the PostInput virtual method. Note that $rivSession stores
# the application object returned in a previous call to the RIV::Agent
# or RIV::App constructor.
$ok = $rivSession->PostInput($tag, $data);


# Call the DecryptPassword and EncryptPassword virtual methods.
$txtPwd = RIV::DecryptPassword($encPwd);
$encPwd = RIV::EncryptPassword($txtPwd);

# Call the RIV::ReadDir function.
$fileListArrayRef = RIV::ReadDir($dirName);
```

```
# Call the Latency and Retry Limit virtual methods. Note the optional
# parameters. Note also that $rivSession stores the application object
# returned in a previous call to the RIV::Agent or RIV::App constructor.
$latency = $rivSession->Latency( [$timeoutMilliSeconds] );
$retryLimit = $rivSession->RetryLimit( [$retryLimit] );

# Call the two versions of the PublishMessage virtual methods. Note that
# $rivSession stores the application object returned in a previous call
# to the RIV::Agent or RIV::App constructor.
$ok = $rivSession->PublishMessage($subject, $message);
$ok = $rivSession->PublishMessage($subject, $refHash);

# Call the AddSubject and AddTimer virtual methods. Note that
# $rivsession stores the application object returned in a previous
# call to the RIV::Agent or RIV::App constructor.
$ok = $rivSession->AddSubject($subject, $tag);
$ok = $rivSession->AddTimer($timerval, $tag, $isRepeat);

# Fetch a row
my $dat = $app->RIV::FetchRow($rsKey)

# Get a result set
$rsKey = $rivSession->RIV::GetResultSet([ $waitTime ]);

# Remove an IO Handle
$ok = $rivSession->RemoveIoHandle($fileHandle, $tag);

# Remove a subject
$ok = $rivSession->RemoveSubject($subject, $isRaw);
```

AddIoHandle

The `AddIoHandle` virtual method adds the file or socket handle to the I/O list.

Virtual Method Synopsis

```
AddIoHandle($fileHandle, $tag)
```

Parameters

$fileHandle
Specifies the file or socket handle.

$tag
Specifies the tag to be appended to `"USER_"` and to be associated with the messages.

Description

The `AddIoHandle` virtual method adds the file or socket handle specified in the *$fileHandle* parameter to the I/O list. When the file descriptor is available for reading, the tag `"USER_$tag"` is returned to the Perl application through a call to the `RIV::GetResult()` function.

Example Usage

```
$app->AddIoHandle(STDOUT, "test");
```

Returns

Upon completion, the `AddIoHandle` virtual method returns:

- 0 (zero) — The attempt to add the file or socket handle to the I/O list was unsuccessful.
- 1 — The attempt to add the file or socket handle to the I/O list was successful.

See Also

- “RIV::Agent module reference” on page 187

- “RIV::App module reference” on page 211

AddSubject

The AddSubject virtual method binds the application to the specified message broker subject.

Virtual Method Synopsis

```
AddSubject($subject, $tag)
```

Parameters

$subject
Specifies the message broker subject to which AddSubject binds the application.

$tag
Specifies the tag to be appended to "USER_".

Description

The AddSubject virtual method:

- Binds the application to the message broker subject specified in the *$subject* parameter.
- Appends the tag specified in the *$tag* parameter to "USER_". The "USER_*$tag*" messages are returned in a call to the RIV::GetResult() function.

The subject is automatically appended with the domain in order to limit messages to purely those for the current domain.

Example Usage

The following example assumes a previous call to the RIV::App constructor, which returns a client/server application object to $app. A call could also be made to the RIV::Agent constructor, which returns a discovery agent application object (typically, to $agent).

```
$ok = $app->AddSubject('ITNM/MODEL/NOTIFY', 'model');
```

Returns

Upon completion, the AddSubject virtual method returns:

- 0 (zero) — The attempt to bind the application to the message broker subject was unsuccessful.
- 1 — The attempt to bind the application to the message broker subject was successful.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::RivDebug” on page 185
- “RIV::GetResult” on page 178

AddTimer

The AddTimer virtual method creates a single-shot or repeating timer.

Virtual Method Synopsis

```
AddTimer($timerVal, $tag, $isRepeat)
```

Parameters

$timerVal
Specifies the time interval, in milliseconds, between timer events.

$tag
Specifies the tag to be appended to "USER_".

$isRepeat
Specifies the type of timer to create. The value 0 (zero) creates a single-shot timer and the value 1 creates a repeating timer.

Description

The AddIimer virtual method:

- Creates a single-shot or repeating timer, depending on the value passed to the *$isRepeat* parameter. The value of the *$timerVal* parameter specifies the interval, in milliseconds, between the timer events.
- Appends the tag specified in the *$tag* parameter to "USER_". The timer events are returned to the Perl application as "USER_$tag" messages through a call to the RIV::GetResult function.

Example Usage

The following example assumes a previous call to the RIV::App constructor, which returns a client/server application object to $app. A call could also be made to the RIV::Agent constructor, which returns a discovery agent application object (typically, to $agent).

```
$ok = $app->AddTimer(100, "TIMER", 1);
```

Returns

Upon completion, the AddTimer virtual method returns:

- 0 (zero) — The attempt to create a single-shot or repeating timer was unsuccessful.
- 1 — The attempt to create a single-shot or repeating timer was successful.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212
- "RIV::RivDebug" on page 185
- "RIV::GetResult" on page 178

DebugLevel

The RIV module provides access to the global Network Manager debugging level setting through the `$RIV::DebugLevel` variable.

Variable Synopsis

```
$RIV::DebugLevel
```

Description

The RIV module provides access to the global Network Manager debugging level setting through the `$RIV::DebugLevel` variable. Changing the value of this variable will affect all debugging output. The default value is 0 (zero).

Typically, you use this variable with the following RIV module function:

- `RIV::RivDebug`

See Also

- "RIV::RivDebug" on page 185

DecryptPassword

The `DecryptPassword` virtual method decrypts the specified encrypted password.

Synopsis

```
DecryptPassword($encPwd)
```

Parameters

$encPwd
Specifies the encrypted password to be decrypted.

Description

The `DecryptPassword` virtual method decrypts the encrypted password specified in the *$encPwd* parameter. You previously encrypted this password in a call to the `EncryptPassword` virtual method. Note that the encryption key must be the same as that used to encrypt the original password.

Example Usage

```
$txtPwd = RIV::DecryptPassword($encPwd);
```

Returns

Upon completion, the `DecryptPassword` virtual method returns the plain text password or `undef` if an error occurred.

See Also

- "EncryptPassword" on page 170
- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212

EncryptPassword

The `EncryptPassword` virtual method returns an encrypted representation of the specified password.

Synopsis

```
EncryptPassword($txtPwd)
```

Parameters

$txtPwd
Specifies the plain text password to be encrypted.

Description

The `EncryptPassword` virtual method encrypts the plain text password specified in the *$txtPwd* parameter.

Example Usage

```
$encPwd = RIV::EncryptPassword($txtPwd);
```

Returns

Upon completion, the `EncryptPassword` virtual method returns an encrypted and ASCII encoded representation of the specified plain text password or `undef` if an error occurred.

See Also

- “DecryptPassword” on page 169
- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

Latency

The `Latency` virtual method sets or retrieves the timeout for queries.

Virtual Method Synopsis

```
Latency([$timeoutMilliseconds])
```

Parameters

$timeoutMilliseconds
Specifies the timeout, in milliseconds, for the queries. This is an optional parameter and if omitted (that is, no timeout is specified) it returns the value of the timeout.

Description

The `Latency` virtual method:

- Sets a timeout, in milliseconds, for queries if a timeout value is passed to the *$timeoutMilliseconds* parameter. The value passed to *$timeoutMilliseconds* cannot be the `undef` value.
- Returns the timeout, in milliseconds, for queries if the *$timeoutMilliseconds* parameter is omitted (that is, no timeout is specified). If an acknowledgement is not received within this time, further requests are

sent up to the retry limit. (The retry limit is specified in a call to the `RetryLimit` virtual method). If no acknowledgement is received after (retry*timeout), an error is returned to the caller.

Example Usage

The following examples assume a previous call to the `RIV::App` constructor, which returns a client/server application object to $app. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

The following example shows a call with no *$timeoutMilliseconds* parameter specified:

```
$latency = $app->Latency();
```

The following example shows a call with the *$timeoutMilliseconds* parameter specified:

```
$app->Latency(1000);
```

Returns

Upon completion, the `Latency` virtual method returns the timeout, in milliseconds, if the *$timeoutMilliseconds* parameter is omitted.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RetryLimit” on page 175
- “RIV::RivDebug” on page 185

MessageLevel

The RIV module provides access to the global Network Manager logging level setting through the `$RIV::MessageLevel` variable.

Variable Synopsis

```
$RIV::MessageLevel
```

Description

The RIV module provides access to the global Network Manager logging setting through the `$RIV::MessageLevel` variable. Changing the value of this variable will affect all logging output.

Typically, you use this variable with the following RIV module function:

- `RIV::RivMessage`

See Also

- “RIV::RivMessage” on page 187

PostInput

The PostInput virtual method adds a message to the queue.

Virtual Method Synopsis

```
PostInput($tag, $data)
```

Parameters

$tag
Specifies the tag to be associated with the input message.

$data
Specifies the data in the message.

Description

The PostInput virtual method adds the input message specified in the *$data* parameter to the queue along with the associated tag specified in the *$tag* parameter.

Example Usage

The following example assumes a previous call to the RIV::App constructor, which returns a client/server application object to $app. A call could also be made to the RIV::Agent constructor, which returns a discovery agent application object (typically, to $agent).

```
$app = RIV::App::new(.......);
$app->PostInput("myTag", "test data");
```

Returns

Upon completion, the PostInput virtual method returns:

- 0 (zero) — The attempt to add the input message to the queue was unsuccessful.
- 1 — The attempt to add the input message to the queue was successful.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::RivDebug” on page 185

PublishMessage

The PublishMessage virtual method publishes the specified message string.

Virtual Method Synopsis

```
PublishMessage($subject, $message)
```

Parameters

$subject
Specifies the unqualified Network Manager subject used for message broker messaging.

$message
Specifies a valid ASCII message string.

Description

The PublishMessage virtual method publishes the message string specified in the *$message* parameter on the subject specified in the *$subject* parameter. The value specified in *$subject* must be unqualified, that is, it must be without the **.DOMAIN** suffix. The message in *$message* must be a valid ASCII string.

Example Usage

The following example assumes a previous call to the RIV::App constructor, which returns a client/server application object to $app. A call could also be made to the RIV::Agent constructor, which returns a discovery agent application object (typically, to $agent).

```
$ok = $app->PublishMessage('ITNM/MODEL/QUERY', "hello");
```

Returns

Upon completion, the PublishMessage virtual method returns:

- 0 (zero) — The attempt to publish the specified message was unsuccessful.
- 1 — The attempt to publish the specified message was successful.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::RivDebug" on page 185
- "PublishMessage" on page 173

PublishMessage

The PublishMessage virtual method encodes the hash reference into a message broker message.

Virtual Method Synopsis

```
PublishMessage($subject, $refHash)
```

Parameters

$subject
Specifies the unqualified Network Manager subject used for message broker messaging.

$refHash
Specifies a reference to a hash that contains the message to be sent.

Description

The PublishMessage virtual method encodes the hash specified in the $refHash parameter into a message broker message and publishes it on the subject specified in the $subject parameter. The hash passed to $refHash may be nested. The value passed to the $subject parameter must be unqualified, that is, it must be without the **.DOMAIN** suffix.

Note: The message type and contents must be what the consumer is expecting. There is a chance that the core processes will SIGSEGV if they receive unexpected data (for example, publishing a string message to a NOTIFY subject).

Example Usage

The following example assumes a previous call to the RIV::App constructor, which returns a client/server application object to $app. A call could also be made to the RIV::Agent constructor, which returns a discovery agent application object (typically, to $agent).

```
my %gg; $gg{'foo'} = 'bar'; $gg{'color'} = 'red';
$ok = $app->PublishMessage('ITNM/MODEL/QUERY', \%gg);
```

Returns

Upon completion, the PublishMessage virtual method returns:

- 0 (zero) — The attempt to encode the hash and publish the specified message was unsuccessful.
- 1 — The attempt to encode the hash and publish the specified message was successful.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::RivDebug" on page 185
- "PublishMessage" on page 172

RemoveIoHandle

The RemoveIoHandle virtual method removes the file handle $fielHandle from the I/O list.

Virtual Method Synopsis

```
RemoveIoHandle($fileHandle, $tag)
```

Parameters

$fileHandle
Specifies the file or socket handle to be removed from the I/O list.

$tag
Specifies the tag to be associated with the messages.

Description

The RemoveIoHandle virtual method removes the file or socket handle that is specified in the *$fileHandle* parameter from the I/O list.

Example Usage

```
$app->RemoveIoHandle(STDOUT "test");
```

Returns

Upon completion, the RemoveIoHandle virtual method returns:

- 0 (zero) — The attempt to remove the file or socket handle from the I/O list was unsuccessful.
- 1 — The attempt to remove the file or socket handle from the I/O list was successful.

See Also

- “RIV::Agent module reference” on page 187
- “RIV::App module reference” on page 211

RemoveSubject

The `RemoveSubject` virtual method removes the listener on the application to the Message Broker subject `$subject`.

Virtual Method Synopsis

```
RemoveSubject($subject, $isRaw)
```

Parameters

$subject
Specifies the Message Broker subject from which to remove the listener.

$isRaw
Specifies whether the domain is appended to subject, or not.

- `0` — Add the domain to the subject.
- 1 — Do not add the domain to the subject.

Description

The `RemoveSubject` virtual method removes the listener on the application to the Message Broker subject `$subject`. If `$isRaw = 0` then the domain is appended to the subject to be removed.

Example Usage

```
$ok =$app->RIV::RemoveSubject('ITNM/MODEL/NOTIFY', 0);
```

Returns

Upon completion, the `RemoveSubject` virtual method returns:

- 0 (zero) — The attempt to remove the application from the subject was unsuccessful.
- 1 — The attempt to remove the application from the subject was successful.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

RetryLimit

The `RetryLimit` virtual method sets the retry limit for queries or returns the maximum number of retries for queries.

Virtual Method Synopsis

```
RetryLimit([$retryLimit])
```

Parameters

$retryLimit
Specifies the retry limit for a specified query. This is an optional parameter and if omitted (that is, no retry limit is specified) it returns the maximum number of retries.

Description

The `RetryLimit` virtual method:

- Sets a retry limit for queries if a value is passed to the *$retryLimit* parameter.
- Returns the maximum number of retries for a specified query if the *$retryLimit* parameter is omitted (that is, no retry limit is specified).

Example Usage

The following examples assume a previous call to the `RIV::App` constructor, which returns a client/server application object to `$app`. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

The following usage example shows a call with no *$retryLimit* parameter specified:

```
$retry = $app->RetryLimit();
```

The following usage example shows a call with the *$retryLimit* parameter specified:

```
$app->RetryLimit(5);
```

Returns

Upon completion, the `RetryLimit` virtual method returns the maximum number of retries for a specified query if the *$retryLimit* parameter is omitted.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::RivDebug” on page 185

RIV::FetchRow

The `RIV::FetchRow` method is used to fetch the next row from a results set retrieved by `RIV::GetResultSet()` and returns `undef` if there are no more rows to return.

Synopsis

```
RIV::FetchRow([$rsKey])
```

Parameters

$rsKey
The results set key that was returned by the `RIV::GetResultSet()` method.

Description

The `RIV::FetchRow` method is used to fetch the next row from a results set retrieved by `RIV::GetResultSet()` and returns `undef` if there are no more rows to return. Care must be taken to iterate over all rows in the results set (even if the data is not required) otherwise the memory that is allocated to hold remainder of the results set is not freed.

Example Usage

```
my $isSelect = 1;
$oql->Send($request, $isSelect);
my $rsKey = $app->RIV::GetResultSet();

if ($rsKey)
{
  while (my $dat = $app->RIV::FetchRow($rsKey))
  {
    # process data
  }
}
```

Returns

Upon completion, the `RIV::FetchRow` method returns a Perl hash that represents the next row of the results set, or `undef` if there is no more data.

See Also

- "RIV::Agent module reference" on page 187
- "RIV::App module reference" on page 211
- "RIV::GetResultSet" on page 180

RIV::GetInput

The `RIV::GetInput` function has been deprecated by `RIV::GetResult`. The documentation exists for historical purposes only.

Synopsis

```
RIV::GetInput($waitTime [,$pattern])
```

Parameters

$waitTime
Specifies the time, in seconds, to wait before returning. If *$waitTime* is negative, `RIV::GetInput` waits forever for the response.

$pattern
Specifies the pattern of tags that the user is interested in. Only messages with a matching tag will be returned. All other messages will be left in the queue for retrieval at a later time. This parameter is optional.

Description

The `RIV::GetInput` function provides a mechanism for synchronizing a single-threaded Perl application with the multithreaded Network Manager platform. There is currently no support for direct interface with Perl threads.

The normal usage of `RIV::GetInput` takes a single parameter to specify the number of seconds to wait for input before returning. A value of 0 (zero) means "do not wait", and a negative value means "wait forever".

Because Network Manager platform receives input either directly or indirectly through message broker, the input data is placed on a FIFO together with its identifying tag. One input item is returned for each call to `RIV::GetInput`. Items are returned as an array of size 3 containing the item tag, item tag value, and the application domain (that is, the domain string specified in the call to `RIV::App::new`). For example:

```
($tag, $data, $domain) = RIV::GetInput(-1);
```

If there is only one active RIV::App, the domain value may be ignored. However, if multiple RIV::App objects have been created, the value of *$domain* must be used to determine the source of the input.

Value types depend on the item returned and must be interpreted in the context of the value of *$tag*. Tag values are either specified in a call to create the input stream or are from a set of standard tags. User specified tags are returned from RIV::GetInput with the prefix USER_. Standard tags include:

- QUERY — (RIV::OQL query results)
- UPDATE — (RIV::OQL updates)
- NE — (RIV::Agent)
- TIMEOUT — (all - wait time exceeded and no data)

The extended form of RIV::GetInput uses a second parameter (*$pattern*) to specify a regular expression pattern for matching against input tag strings. Only data items with matching tags will be returned. This form is useful for temporarily suspending delivery of input to all but the wanted channel and has the effect of taking input data items out of turn. Non-matching input tags are kept in the queue and will be delivered in sequence when the standard form of RIV::GetInput is used, or a matching pattern is specified to a subsequent call of the extended version.

Example Usage

```
($tag, $data, $domain) = RIV::GetInput(-1);
```

Returns

Upon successful completion, the RIV::GetInput function returns:

- *$tag* — The tag associated with the message.
- *$data* — The data associated with the tag. The *$data* value could be a string or a reference to any data structure and will be interpreted based on the *$tag* value.
- *$domain* — The domain name. This return will only be of interest if multiple domains are running.

See Also

- “RIV::GetResult” on page 178

RIV::GetResult

The RIV::GetResult function provides the "standard" mechanism for synchronizing a single-threaded Perl application with the multi-threaded Network Manager platform. There is currently no support for a direct interface with Perl threads.

Synopsis

```
RIV::GetResult([$waitTime])
```

Parameters

$waitTime
Specifies the time, in seconds, to wait for input before returning. If *$waitTime* is negative, RIV::GetResult waits forever for the response. This is an optional parameter that defaults to the latency of the application.

Description

The typical call to the RIV::GetResult function takes a single parameter to specify the number of seconds to wait for input before returning. A value of 0 (zero) means "do not wait" and a value of minus

one (-1) means "wait forever". The *$waitTime* parameter is optional and, if it is not specified, it defaults to the latency associated with the application.

As the Network Manager platform receives input (either directly or indirectly through message broker), the input data is placed on a FIFO basis, together with its identifying tag. One input item is returned for each call to `RIV::GetResult`. Items are returned as an array of size 3, containing the item tag, its value and the application domain (that is, the domain string specified in a call to the `RIV::App::new()` constructor). For example:

```
my ($tag, $data, $domain) = $app->RIV::GetResult(-1);
```

If there is only one active `RIV::App`, the domain value may be ignored. However, if multiple `RIV::App` objects have been created, the value of *$domain* must be used to determine the source of the input.

Value types depend on the item returned and must be interpreted in the context of the value of *$tag*. Tag values are either specified in a call to create the input stream or are from a set of standard tags. User specified tags are returned from `RIV::GetResult()` with the prefix "USER_". The following example identifies the standard tags:

```
OQLQuery  (RIV::OQL query results)
OQLUpdate (RIV::OQL updates)
NE        (RIV::Agent)
TIMEOUT   (all - wait time exceeded and no data)
```

Example Usage

The following example assumes a previous call to the `RIV::App` constructor, which returns a client/server application object to $app. A call could also be made to the `RIV::Agent` constructor, which returns a discovery agent application object (typically, to `$agent`).

```
($tag, $data, $domain) = $app->RIV::GetResult();
($tag, $data, $domain) = $app->RIV::GetResult(10);
($tag, $data, $domain) = $app->RIV::GetResult(-1);
```

Returns

Upon successful completion, the `RIV::GetResult` function returns:

- *$tag* — The tag associated with the message.
- *$data* — The data associated with the tag. The *$data* value could be a string or a reference to any data structure and will be interpreted based on the *$tag* value.
- *$domain* — The domain name. This return will only be of interest if multiple domains are running.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::GetInput” on page 177
- “RIV::RivDebug” on page 185

RIV::GetResultSet

This RIV::GetResultSet method is much like RIV::GetResult() . However, instead of returning the entire results set as a Perl hash the RIV::GetResultSet method returns a key that can be used as an argument to RIV::FetchRow() to retrieve the results set one row at a time.

Synopsis

```
RIV::GetResultSet([$waitTime])
```

Parameters

$waitTime
Specifies the time, in seconds, to wait for input before it returns information. If $waitTime is negative, RIV::GetResult waits forever for the response. This parameter is an optional parameter that defaults to the latency of the application.

Description

Use the RIV::GetResultSet method to retrieve data from tables with large numbers of rows as the RIV::GetResultSet method uses less memory than the RIV::GetResult() alternative. The results set is cached and can be accessed by using only the key that is returned by this method. Therefore, care must be taken to iterate over every row of the results set, even if the value is not required. Otherwise, the memory that is used by the results set cannot be reclaimed by the script.

Example Usage

```
my $isSelect = 1;
$oql->Send($request, $isSelect);
my $rsKey = $app->RIV::GetResultSet();

if ($rsKey)
{
  while (my $dat = $app->RIV::FetchRow($rsKey))
  {
    # process data
  }
}
```

Returns

Upon completion, the RIV::GetResultSet method returns a scalar results set key that is passed to RIV::FetchRow() to access the data within the results set.

See Also

- “RIV::Agent module reference” on page 187
- “RIV::App module reference” on page 211
- “RIV::GetResult” on page 178

RIV::InputFilter

The RIV::InputFilter function binds the function referenced by *$function* to input tags matching the regular expression *$pattern*.

Synopsis

```
RIV::InputFilter($pattern [,$function]
[,$arg])
```

Parameters

$pattern
Specifies the tag corresponding to the regular expression to which the filter must be run.

$function
Specifies the function that will be executed if the input tag matches the regular expression passed to the *$pattern* parameter. This parameter is optional. If no function is specified, the existing callback for the pattern is deleted.

$arg
Specifies an argument to be passed to the function specified in the *$function* parameter. This parameter is optional.

Description

The `RIV::InputFilter` function binds the function referenced by *$function* to input tags matching the regular expression *$pattern*. Whenever the application program calls the `RIV::GetResult` function and data with a matching tag is returned, the corresponding function is called instead of a return from `RIV::GetResult`. If all input tags match one of the patterns passed to `RIV::InputFilter`, the effect is as if the original call to `RIV::GetResult` never returned. The called function must not call `RIV::GetResult`. Calling the `RIV::InputFilter` function with a value of `undef` for *$function* removes the filter.

Example Usage

```
$ok = RIV::InputFilter("NE", $method1);
```

Returns

Upon completion, the `RIV::InputFilter` function returns:

- 0 (zero) — The attempt to bind the function referenced by *$function* was unsuccessful.
- 1 — The attempt to bind the function referenced by *$function* was successful.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::GetResult” on page 178
- “RIV::RivDebug” on page 185

RIV::InputQueueLength

The `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue.

Synopsis

```
RIV::InputQueueLength()
```

Parameters

None

Description

The `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue, that is, the number of times `RIV::GetResult` would need to be called in order to drain the queue.

Example Usage

```
$queue_length = RIV::InputQueueLength();
```

Returns

Upon successful completion, the `RIV::InputQueueLength` function returns the number of items waiting in the application's input queue.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212
- "RIV::GetResult" on page 178
- "RIV::RivDebug" on page 185

RIV::IsIpNotLoopBackOrMulticast

The `RIV::IsIpNotLoopBackOrMulticast` function returns true if the address parameter is a valid IP address, and not a loop back or multicast address.

Synopsis

```
RIV::IsIpNotLoopBackOrMulticast($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpNotLoopBackOrMulticast` function returns true if the address passed to the *$ipAddress* parameter is a valid IP address, and not a loop back or multicast address.

Example Usage

```
$result = RIV::IsIpNotLoopBackOrMulticast($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpNotLoopBackOrMulticast` function returns:

- 0 (zero) — The IP address is not valid or the IP address is a loop back or multicast address.
- 1 — The IP address is valid.

See Also

- "RIV::Agent Constructor" on page 188

- “RIV::App Constructor” on page 212
- “RIV::RivDebug” on page 185

RIV::IsIpValid

The `RIV::IsIpValid` function returns true if the address parameter is a valid IP address.

Synopsis

```
RIV::IsIpValid($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpValid` function returns true if the address passed to the *$ipAddress* parameter is a valid IP address. More specifically, the function checks if *$ipAddress* is a valid IPv4 or IPv6 address using the functions specific to those address families.

Example Usage

```
$result = RIV::IsIpValid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpValid` function returns:

- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::IsIpv4Valid” on page 183
- “RIV::IsIpv6Valid” on page 184
- “RIV::RivDebug” on page 185

RIV::IsIpv4Valid

The `RIV::IsIpv4Valid` function returns true if the address parameter is a valid IPv4 address.

Synopsis

```
RIV::IsIpv4Valid($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpv4Valid` function returns true if the address passed to the *$ipAddress* parameter is a valid IPv4 address. More specifically, the function checks that the IP address is of the form `a.b.c.d` and that each number in the IP address (`a, b, c, d`) is less than 256.

Example Usage

```
$result = RIV::IsIpv4Valid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpv4Valid` function returns:

- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212
- "RIV::IsIpValid" on page 183
- "RIV::IsIpv6Valid" on page 184
- "RIV::RivDebug" on page 185

RIV::IsIpv6Valid

The `RIV::IsIpv6Valid` function returns true if the address parameter is a valid IPv6 address.

Synopsis

```
RIV::IsIpv6Valid($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address that must be checked for validity.

Description

The `RIV::IsIpv6Valid` function returns true if the address passed to the *$ipAddress* parameter is a valid IPv6 address. More specifically, the function checks that the IP address is of the standard forms as defined in RFC429.

Example Usage

```
$result = RIV::IsIpv6Valid($ipAddress);
```

Returns

Upon completion, the `RIV::IsIpv6Valid` function returns:

- 0 (zero) — The IP address is not valid.
- 1 — The IP address is valid.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212
- “RIV::IsIpValid” on page 183
- “RIV::IsIpv4Valid” on page 183
- “RIV::RivDebug” on page 185

RIV::ReadDir

The `RIV::ReadDir` function returns a reference to an array of filenames that reside in the specified directory.

Synopsis

```
RIV::ReadDir($dirName)
```

Parameters

$dirName
Specifies the name of the directory to read.

Description

The `RIV::ReadDir` function returns a reference to an array of filenames that reside in the directory specified in the *$dirName* parameter. The `RIV::ReadDir` function provides the same functionality as the standard Perl `readdir` function. The `RIV::ReadDir` function is supplied to accommodate known issues when trying to use `readdir` with **ncp_perl** on some Linux® platforms.

Example Usage

```
$fileListArrayRef = RIV::ReadDir($dirName);
```

Returns

Upon completion, the `RIV::ReadDir` function returns a reference to an array of filenames that reside in the directory specified in the *$dirName* parameter.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

RIV::RivDebug

The `RIV::RivDebug` function prints a list of debug message strings to the standard output.

Synopsis

```
RIV::RivDebug($lvl,@debugMessageStrings)
```

Parameters

$lvl
Specifies the debug level. Specify a value of 1-4, where 4 represents the most detailed output.

@debugMessageStrings
Specifies the strings to be printed when the debug level is set.

Description

The RIV::RivDebug function prints the space-concatenated list of strings from the *@debugMessageStrings* parameter to the standard output if the value of the $RIV::DebugLevel global variable is equal to, or greater than, the value specified in the *$lvl* parameter.

Example Usage

```
RIV::RivDebug(4, "my debug message here");
```

Returns

Upon completion, the RIV::RivDebug function returns no records or values.

See Also

- "DebugLevel" on page 169

RIV::RivError

The RIV::RivError function provides a convenient way to display error messages.

Synopsis

```
RIV::RivError($class, @errorMessageStrings)
```

Parameters

$class
Specifies the name of the calling Perl API module (for example, RIV::App).

@errorMessageStrings
Specifies the error message string to be printed when an error occurs.

Description

The RIV::RivError function prints the error messages tagged with the name of the calling class (RIV::*) and will integrate with the forthcoming trace package.

Example Usage

```
RIV::RivError("RIV::App", "An error has occurred");
```

Returns

Upon completion, the RIV::RivError function returns no records or values.

See Also

- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212
- "RIV::RivDebug" on page 185

RIV::RivMessage

The `RIV::RivMessage` function prints a list of log message strings to the standard output.

Synopsis

```
RIV::RivMessage($msglvl,@messageLevelStrings)
```

Parameters

$msglvl
Specifies the level of messages to be logged (the default is warn):

- debug
- info
- warn
- error
- fatal

@messageLevelStrings
Specifies the strings to be printed when the logging level is set.

Description

The `RIV::RivMessage` function prints the space-concatenated list of strings from the *@messageLevelStrings* parameter to the standard output if the value of the `$RIV::MessageLevel` global variable is equal to, or greater than, the value specified in the *$msglvl* parameter.

Example Usage

```
RIV::RivMessage("warn", "my messagelevel message here");
```

Returns

Upon completion, the `RIV::RivMessage` function returns no records or values.

See Also

- "MessageLevel" on page 171

RIV::Agent module reference

The `RIV::Agent` module enables developers to implement Network Manager discovery agents.

The `RIV::Agent` module provides a constructor and the following categories of methods:

- SNMP operation methods
- DNS operation methods
- Ping operation methods
- IP and MAC address operation methods
- Telnet operation methods
- Network entity operation methods
- Threads methods
- Collector communication operation methods (also referred to as XML-RPC operation methods)

RIV::Agent module synopsis

The RIV::Agent module synopsis provides summary synopses of the constructor and methods that discovery agents use.

```
# Load the RIV::Agent module
use RIV::Agent;

# Call the RIV::Agent constructor and return a RIV::Agent object
$agent = RIV::Agent::new($param, $agentName);

# Call the SNMP operation methods
$varOp = $agent->SnmpGet($ne, $oid, $instance, $communitySuffix);
$varOpArray = $agent->SnmpGetNext($ne, $oid, $instance, $communitySuffix);
$varOpArray = $agent->SnmpGetBulk($ne, $oidList, $nonRepeaters,
                                  $maxRepeaters, $communitySuffix);

# Call the DNS operation methods
$refAllIpAddrs = $agent->GetDNSAllIpAddrs($name);
$refAllNames = $agent->GetDNSAllNames($ipAddress);
$ip = $agent->GetDNSFirstIpAddr($name);
$name = $agent->GetDNSFirstName($ipAddress);

# Call the IP and MAC address operation methods
$ip = $agent->GetIpArp($macAddress);
$mac = $agent->GetMacArp($ipAddress);
$routelist = $agent->GetTraceRoute($ipAddress, $protocol);

# Call the ping operation methods
$reply = $agent->GetPingIP($ipAddress, $protocol);
$replylist = $agent->GetPingList($ipAddressList, $protocol);
$reply = $agent->GetPingSubnet($subnet, $netMask, $protocol);
$agent->PingIP($ipAddress, $protocol);
$agent->PingList($ipAddressList, $protocol);
$agent->PingSubnet($subnet, $netMask, $protocol);

# Call the Telnet operation methods
$telarray = $agent->GetMultTelnet($ne, $commandList);
$teldata = $agent->GetTelnet($ne, $command, $regExp);
$teldata = $agent->GetTelnetCols($ne, $command, $regExpList, $colNameList);
$telData = $agent->ExtGetTelnet($ne, $commandList, $accessCredentials);
# Call the Collector communication operation methods

# Call an XML-RPC Collector method
$xmlResponse = $agent->GetXMLRPCData($host, $port, "GetDeviceInfo",
$idAgr, $deviceIdArg);
# or ..
$xmlResponse = $agent->GetXMLRPCEntityData($ne, "GetDeviceInfo", $idAgr,
$deviceIdArg);

# Call the Network Entity operation methods
$agent->SendNEToDisco($NE);
$agent->SendNEToNextPhase($NE);

# Call the threads operation methods
$agent->LockThreads();
$agent->UnLockThreads();
```

RIV::Agent Constructor

The RIV::Agent constructor creates a Network Manager discovery agent with the specified name.

Constructor

```
new($param, $agentName)
```

Parameters

$param

Specifies a RIV::Param object that was returned in a previous call to the RIV::Param constructor.

$agentName
Specifies a string that identifies the name of the discovery agent to be created in the domain specified by the RIV::Param object passed to the *$param* parameter.

Description

The RIV::Agent constructor creates a Network Manager discovery agent with an agent name as specified in the *$agentName* parameter. This agent name resides in the domain as specified by the *$param* parameter (that is, a RIV::Param object).

The RIV::Agent constructor uses the Transmission Control Protocol (TCP) to establish the necessary connections to the Discovery Server and Helper Server. To ensure that the databases for the discovery agent are created inside the Discovery Server, the $agentName.agnt file must be defined in the $NCHOME/disco/agents directory before the Discovery Engine executable, ncp_disco, is started.

Example Usage

The following example:

- Calls the RIV::Param constructor and stores the return value (a RIV::Param object) in the *$param* variable.
- Calls the RIV::Agent constructor and specifies a discovery agent name of foo_perl_disco_agent.
- Stores the return value (a RIV::Agent object) in the *$agent* variable.

```
$param = new RIV::Param();
$agent = new RIV::Agent($param, "foo_perl_disco_agent");
```

Returns

Upon completion, the RIV::Agent constructor returns a RIV::Agent object. This object is associated with the discovery agent specified in the *$agentName* parameter.

ExtGetTelnet

The ExtGetTelnet method is used to gather device data via telnet, rather than SNMP.

Method Synopsis

```
ExtGetTelnet($ne, $commandList, $accessCredentials)
```

Parameters

$ne
The device to issue the command on.

$commandList
An array of hashes that contain the extended commands.

$accessCredentials
An optional hash that contains the access credentials.

Description

The ExtGetTelnet method is used to gather device data via telnet, rather than SNMP.

Notes

The GetTelnet method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Example Usage

The following example;

- Assumes a previous call to the `RIV::Agent constructor`, which returns a `RIV::Agent object` that is represented by `$agent->`.
- `$host` specifies the device to issue the command on.
- `$command` specifies the commands to run.
- `$accessCredential` specifies the access credentials.
- Returns a hash that contains the results of the telnet command.
- Calls the Dumper method to print the results to standard output.

```
my $host = { "m_IpAddress" => "172.20.1.5",
             "m_ObjectId" => "1.3.6.1.4.1.9.1.222",
             "m_ResponseTime" => 1000 };
my $commands = [{"m_Command" => "show clock"}];
my $accessCredentials = { "m_Username" => "user",
                          "m_Password" => "password",
                          "m_SSHSupport" => 1 };


my $result = $agent->ExtGetTelnet($host, $commands, $accessCredentials);

print Dumper($result);
```

Returns

Upon completion of the `ExtGetTelnet` method, an array of hashes from each telnet response is returned.

GetDNSAllIpAddrs

The `GetDNSAllIpAddrs` method returns all IP addresses corresponding to a particular node name.

Method Synopsis

```
GetDNSAllIpAddrs($name)
```

Parameters

$name

Specifies the name of the node whose corresponding IP addresses are of interest.

Description

The `GetDNSAllIpAddrs` method returns all IP addresses corresponding to the node name specified in the *$name* parameter.

Notes

The `GetDNSAllIpAddrs` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).

- Returns to the *$refAllIpAddrs* variable a reference to an array that contains all IP addresses corresponding to the node called `foo`.
- Calls the `print` operator to send each IP address in the list to standard output.

```
$refAllIpAddrs = $agent->GetDNSAllIpAddrs("foo");
print @$refAllIpAddrs;
```

Returns

Upon completion, the `GetDNSAllIpAddrs` method returns a reference to an array of IP addresses corresponding to the specified node name.

GetDNSAllNames

The `GetDNSAllNames` method returns all node names corresponding to a specific IP address.

Method Synopsis

```
GetDNSAllNames($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address whose corresponding node names are of interest.

Description

The `GetDNSAllNames` method returns all node names corresponding to the IP address specified in the *$ipAddress* parameter. by issuing a DNS request to the Helper Server.

Notes

The `GetDNSAllNames` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$refAllNames* variable a reference to an array that contains all node names corresponding to the IP address `1.2.3.4`.
- Calls the `print` operator to send each node name in the list to standard output.

```
$refAllNames = $agent->GetDNSAllNames("1.2.3.4");
print @$refAllNames;
```

Returns

Upon completion, the `GetDNSAllNames` method returns a reference to an array of names corresponding to a specific IP address.

GetDNSFirstIpAddr

The `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses for the specified node.

Method Synopsis

```
GetDNSFirstIpAddr($name)
```

Parameters

$name
Specifies the name of the node whose first IP address in the corresponding list of IP addresses is of interest.

Description

The `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses corresponding to the node specified in the *$name* parameter.

Notes

The `GetDNSFirstIpAddr` method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$ip* variable the first IP address in the list of IP addresses corresponding to the node called `foo`.

```
$ip = $agent->GetDNSFirstIpAddr("foo");
```

Returns

Upon completion, the `GetDNSFirstIpAddr` method returns the first IP address in the list of IP addresses for the specified node. This IP address is a scalar value.

GetDNSFirstName

The `GetDNSFirstName` method returns the first node name in the list of node names for the specified IP address.

Method Synopsis

```
GetDNSFirstName($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address whose first node name in the corresponding list of node names is of interest.

Description

The GetDNSFirstName method returns the first node name in the list of node names corresponding to the IP address specified in the *$ipAddress* parameter.

Notes

The GetDNSFirstName method issues the appropriate DNS request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate DNS request.

Example Usage

The following example:

- Assumes a previous call to the RIV::Agent constructor, which returns a RIV::Agent object (represented by $agent->).
- Returns to the *$name* variable the first node name in the list of node names corresponding to the IP address 1.2.3.1.

```
$name = $agent->GetDNSFirstName("1.2.3.1");
```

Returns

Upon completion, the GetDNSFirstName method returns the first node name in the list of node names for the specified IP address. This node name is a scalar value.

GetIpArp

The GetIpArp method converts the specified MAC address to its corresponding IP address.

Method Synopsis

```
GetIpArp($macAddress)
```

Parameters

$macAddress
Specifies the MAC address to be converted to its corresponding IP address.

Description

The GetIpArp method converts the MAC address specified in the *macAddress* parameter to its corresponding IP address.

Notes

The GetIpArp method issues the appropriate ARP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ARP request.

Example Usage

The following example:

- Assumes a previous call to the RIV::Agent constructor, which returns a RIV::Agent object (represented by $agent->).
- Returns to the *$ip* variable the IP address corresponding to the MAC address 00-0C-F1-56-98-AD.

```
$ip = $agent->GetIpArp("00-0C-F1-56-98-AD");
```

Returns

Upon completion, the `GetIpArp` method returns the IP address corresponding to the specified MAC address.

GetMacArp

The `GetMacArp` method converts the specified IP address to a MAC address.

Method Synopsis

```
GetMacArp($ipAddress)
```

Parameters

$ipAddress
Specifies the IP address to be converted to an associated MAC address.

Description

The `GetMacArp` method converts the IP address specified in the *ipAddress* parameter to an associated MAC address.

Notes

The `GetMacArp` method issues the appropriate ARP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ARP request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$macAddress* variable the MAC address corresponding to the IP address `1.2.3.1`.

```
$macAddress = $agent->GetMacArp("1.2.3.1");
```

Returns

Upon completion, the `GetMacArp` method returns the MAC address corresponding to the specified IP address.

GetMultTelnet

The `GetMultTelnet` method initiates a Telnet session on the specified network device and then executes the specified Telnet commands on that network device.

Method Synopsis

```
GetMultTelnet($ne, $commandList)
```

Parameters

$ne
Specifies a reference to the network entity, in this case the network device on which to execute the Telnet commands specified in the `$commandList` parameter.

$commandList
Specifies an array that contains the Telnet commands to execute on the network device specified in the $ne parameter.

Description

The `GetMultTelnet` method initiates a Telnet session on the network device specified in the $ne parameter. It then executes the Telnet commands specified in the `$commandList` parameter on that network device.

Notes

The `GetMultTelnet` method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the `GetMultTelnet` method returns an array of data corresponding to each Telnet command executed during the Telnet session.

GetPingIP

The `GetPingIP` method issues a ping at the specified IP address to determine if a network device exists at that address.

Method Synopsis

```
GetPingIP($ipAddress [, $protocol])
```

Parameters

$ipAddress
Specifies the IP address to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingIP` method pings the IP address specified in the *ipAddress* parameter. A network device that exists at the specified IP address will respond to this ping request.

Notes

The `GetPingIP` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Example Usage

The following example:

- Assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).
- Returns to the *$device_exists* variable a value of 0 (zero) or 1.

```
$device_exists = $agent->GetPingIP("1.2.3.1");
```

Returns

Upon completion, the `GetPingIP` method returns one of the following values:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetPingList

The `GetPingList` method issues a ping at the specified list of IP addresses to determine if network devices exist at those addresses.

Method Synopsis

```
GetPingList($ipAddressList [, $protocol])
```

Parameters

$ipAddressList
Specifies the list of IP addresses to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingList` method pings the list of IP addresses specified in the *ipAddressList* parameter. Network devices that exist at the specified IP addresses will respond to this ping request.

Notes

The `GetPingList` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `GetPingList` method returns a list that identifies whether the network devices exist at the specified IP addresses. The following values are specified in the list:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetPingSubnet

The `GetPingSubnet` method pings the specified subnet and returns whether a reply was received. issues a ping at the specified subnet to determine if one or more network devices exist at that subnet.

Method Synopsis

```
GetPingSubnet($subnet, $netMask [, $protocol])
```

Parameters

$subnet
Specifies the IP address of the subnet to be pinged. Typically, subnets are defined as all devices whose IP addresses have the same prefix. Thus, all devices with IP addresses that start with `1.1.1` would be part of the same subnet.

$netmask
Specifies the mask used to determine the subnet to which an IP address belongs.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `GetPingSubnet` method pings the subnet specified in the *subnet* parameter Network devices that exist at the specified subnet will respond to this ping request.

Notes

The `GetPingSubnet` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `GetPingSubnet` method returns a list that identifies whether the network devices exist at the specified subnet. The following values are specified in the list:

- 0 (zero) — There is no network device at the specified IP.
- 1 — There is a network device at the specified IP address.

GetTelnet

The `GetTelnet` method initiates a Telnet session on the specified network device and executes the specified Telnet command on that network device.

Method Synopsis

```
GetTelnet($ne, $command, $regExp)
```

Parameters

$ne
Specifies a reference to the network entity, in this case the network device on which to execute the Telnet command specified in the $command parameter.

$command
Specifies the Telnet command to execute on the network device specified in the $ne parameter.

$regExp
Specifies the regular expression to apply to the response of the Telnet command specified in the $command parameter.

Description

The GetTelnet method initiates a Telnet session on the network device specified in the $ne parameter. The GetTelnet method then executes the Telnet command specified in the $command parameter on that network device.

Notes

The GetTelnet method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the GetTelnet method returns the data corresponding to the Telnet command executed during the Telnet session. This data must meet the regular expression supplied in the *$regExp* parameter.

GetTelnetCols

The GetTelnetCols method initiates a Telnet session on the specified network device and executes the specified Telnet command on that network device.

Method Synopsis

```
GetTelnetCols($ne, $command, $regExpList, $colNameList)
```

Parameters

$ne
Specifies a reference to the network entity, in this case the network device on which to execute the Telnet command specified in the $command parameter.

$command
Specifies the Telnet command to execute on the network device specified in the $ne parameter.

$regExpList
Specifies an array of regular expressions to apply to the response of the Telnet command specified in the $command parameter.

$colNameList
Specifies an array of table columns.

Description

The GetTelnetCols method:

- Initiates a Telnet session on the network device specified in the $ne parameter by issuing a Telnet request through the Helper Server.
- Executes the Telnet command specified in the $command parameter on that network device.
- Splits data into columns based on the regular expression specified in the $regExpList parameter. This method is particularly suited to responses to Telnet commands that consist of tables.

Notes

The GetTelnetCols method issues the appropriate Telnet request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate Telnet request.

Returns

Upon completion, the GetTelnetCols method returns the data corresponding to the Telnet command executed during the Telnet session. This data must meet the regular expression supplied in the *$regExp* parameter.

GetTraceRoute

The GetTraceRoute method traces a route to the specified destination IP address and returns the network devices that reside on that route.

Method Synopsis

```
GetTraceRoute($ipAddress [, $protocol])
```

Parameters

$ipAddress
Specifies the destination IP address whose route is to be traced.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The GetTraceRoute method traces a route to the destination IP address specified in the *ipAddress* parameter. Network devices that exist at the IP addresses on the route will respond to ping requests.

Notes

The GetTraceRoute method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the GetTraceRoute method returns a list of the devices that reside at IP addresses on the route ending with the destination address specified in the *$ipAddress* parameter.

GetXMLRPCData

The `GetXMLRPCData` method issues an XML-RPC call, through the XML-RPC Helper, on the specified method to the specified host and port.

Method Synopsis

```
GetXMLRPCData($host, $port, $method, $methodArgArray)
```

Parameters

$host
Specifies the host address of the target device supporting XML-RPC calls.

$port
Specifies the port of the target device to which XML-RPC calls should be made.

$method
Specifies the XML-RPC method to call on the target device.

$methodArgArray
Holds the relevant arguments to the selected XML-RPC method call specified in the *$method* parameter.

Description

The `GetXMLRPCData` method uses the XML-RPC Helper to perform the XML-RPC call specified in the *$method* parameter on the target device specified in the *$host* and *$port* parameters. The call can be a custom call or one of the calls defined in the published Network Manager Collector XML Schema.

Notes

The `GetXMLRPCData` method issues the request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate XML-RPC request.

Returns

Upon completion, the `GetXMLRPCData` method returns the XML data with which the target device responded. It is the responsibility of the caller to interpret this XML data.

GetXMLRPCEntityData

The `GetXMLRPCEntityData` method issues an XML-RPC call, through the XML-RPC Helper, on the specified method. The Collector contacted will be that referenced in the supplied standard entity record (that is, the `.despatch` record).

Method Synopsis

```
GetXMLRPCEntityData($ne, $method, $methodArgArray)
```

Parameters

$ne
Specifies a reference to the network entity record as received in the `.despatch` table of the Collector supporting Agents. The target device's host address and port will be extracted from the data `m_CollectorInfo` within this record.

$method
Specifies the XML-RPC method to call on the target device.

$methodArgArray
Holds the relevant arguments to the selected XML-RPC method call specified in the *$method* parameter.

Description

The `GetXMLRPCEntityData` method uses the XML-RPC Helper to perform the XML-RPC call specified in the *$method* parameter on the target device specified in the *$ne* parameter. The call can be a custom call or one of the calls defined in the published Network Manager Collector XML Schema.

Notes

The `GetXMLRPCEntityData` method issues the request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate XML-RPC request.

Returns

Upon completion, the `GetXMLRPCEntityData` method returns the XML data with which the target device responded. It is the responsibility of the caller to interpret this XML data.

LockThreads

The `LockThreads` method acquires a lock that only a single agent thread may hold at any given time.

Method Synopsis

```
LockThreads()
```

Parameters

None

Description

The `LockThreads` method provides a way for a discovery agent to acquire a lock that only a single agent thread may hold at any given time. This means that the code within the locked section is serialized. You should release the lock by calling the `UnLockThreads` method.

Example Usage

The following example shows a call to the `LockThreads` method followed by a call to the `UnLockThreads` method to release the lock. The example assumes a previous call to the `RIV::Agent` constructor, which returns a `RIV::Agent` object (represented by `$agent->`).

```
$agent->LockThreads();

    #
    # Serialised code goes here
    #

    $agent->UnLockThreads();
```

Returns

Upon completion, the `LockThreads` method does not return any values.

PingIP

The PingIP method pings the specified IP address.

Method Synopsis

```
PingIP($ipAddress [, $protocol])
```

Parameters

$ipAddress
Specifies the IP address to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The PingIP method pings the IP address specified in the *ipAddress* parameter. The method returns without waiting for a response from the network device at that address.

Notes

The PingIP method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the PingIP method returns the value 1 to indicate that it successfully pinged the device at the specified address. Otherwise, it returns the value 0 (zero).

PingList

The PingList method pings the specified list of IP addresses.

Method Synopsis

```
PingList($ipAddressList [, $protocol])
```

Parameters

$ipAddressList
Specifies the list of IP addresses to be pinged.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `PingList` method pings the list of IP addresses specified in the *ipAddressList* parameter. The method returns without waiting for responses from the network devices at the list of addresses.

Notes

The `PingList` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `PingList` method returns the value 1 to indicate that it successfully pinged the devices at the specified addresses. Otherwise, it returns the value 0 (zero).

PingSubnet

The `PingSubnet` method pings the specified subnet.

Method Synopsis

```
PingSubnet($subnet, $netMask [, $protocol])
```

Parameters

$subnet
Specifies the IP address of the subnet to be pinged. Typically, subnets are defined as all devices whose IP addresses have the same prefix. Thus, all devices with IP addresses that start with `1.1.1` would be part of the same subnet.

$netmask
Specifies the mask used to determine the subnet to which an IP address belongs.

$protocol
Specifies an optional parameter that identifies the IP protocol. You can specify one of the following values:

- 1 — Specifies Internet Protocol version 4 (IPv4).
- 3 — Specifies Internet Protocol version 6 (IPv6).

Description

The `PingSubnet` method pings the subnet specified in the *subnet* parameter. The method returns without waiting for responses from the network devices that reside on the specified subnet.

Notes

The `PingSubnet` method issues the appropriate ping request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate ping request.

Returns

Upon completion, the `PingSubnet` method returns the value 1 to indicate that it successfully pinged the devices at the specified subnet. Otherwise, it returns the value 0 (zero).

SendNEToDisco

The SendNeToDisco method sends a processed RIV::Record to the returns table of the particular Agent database in DISCO.

Method Synopsis

```
SendNEToDisco($entity, $lastRecTag)
```

Parameters

$entity
Specifies a reference to a hash list that contains the definition of the record to be sent to DISCO. For convenience, the RIV::Record module is such a hash list that provides nested structures for representing local and remote neighbors.

$lastRecTag
Specifies the record for the network entity according to the following values:

- 0 (zero) — Indicates that more records for this network entity are to follow.
- 1 — Indicates the last record for this network.

Note: If you use RIV::Record module objects, this parameter is ignored.

Description

The SendNEToDISC0 method sends a processed *$entity* record object to the returns table of the particular Agent database in DISCO. Typically, the *$entity* parameter is a RIV::Record module object that contains information about local and remote neighbors.

Example Usage

```
$TestNE=new RIV::Record($data);
..
..
..
$agent->SendNEToDisco($TestNE,0);
```

Returns

None

SendNEToNextPhase

The SendNEToNextPhase method is called by discovery agents that accept data during multiple phases of a network discovery operation. These "multi-phased" discovery agents call SendNEToNextPhase when any data processing for a given phase (for example, phase 1) has been completed.

Method Synopsis

```
RIV::Agent::SendNEToNextPhase($entity)
```

Parameters

$entity
Specifies the network entity to be processed and then marked as having been processed for a specific discovery phase.

Description

The SendNEToNextPhase method marks the network entity specified in the *$entity* parameter as having completed processing in the current discovery phase, and it puts the network entity back on the Agent queue ready for processing in the next discovery phase.

Each discovery agent maintains an Agent queue that contains network entities sent to it from the DISCO process. A typical discovery agent processes the network entities on its Agent queue and then calls the SendNEToDisco method to return the processed network entity to the DISCO process.

Unlike a typical discovery agent, a multi-phased discovery agent must allow for the fact that each discovery phase can be hours apart. Therefore, a multi-phased discovery agent must make multiple calls to the SendNEToNextPhase method to put the network entity back on the Agent queue and mark it as ready for processing in the next discovery phase. Once it completes processing of the network entity, the multi-phased discovery agent calls the SendNEToDisco method to send the data back to the DISCO process.

The following is the basic flow for a multi-phased discovery agent:

- The DISCO process sends a record (network entity) that provides details about a device that this phased discovery agent can process.
- The multi-phased discovery agent receives this record.
- When free, the multi-phased discovery agent starts processing the record in discovery phase 1. When processing is complete in discovery phase 1, the multi-phased discovery agent calls SendNEToNextPhase to put the record back on the Agent queue and mark it as ready for processing in the next discovery phase.

  During any of the discovery phases, a multi-phased agent may also be sending multiple data requests to the Helper Server through the GetSnmp and GetTelnet methods that the RIV::Agent module provides.
- When the phase changes, the DISCO process sends out a broadcast indicating that it is proceeding to the next phase (for example, discovery phase 2). When free, the multi-phased discovery agent starts processing the record marked ready for processing in discovery phase 1. When processing is complete in discovery phase 2, the discovery agent calls SendNEToNextPhase to put the record back on the Agent queue and mark it as ready for processing in the next discovery phase.
- When the phase changes, the DISCO process sends out a broadcast indicating that it is proceeding to the next phase (for example, discovery phase 3). When free, the multi-phased discovery agent completes processing of the record marked ready for processing in discovery phase 2 and calls SendNEToDisco to send all of the data back to the DISCO process.

Notes

You invoke the SendNEToNextPhase method on the RIV::Agent object returned in a previous call to the RIV::Agent constructor. For example:

```
.
.
.
my $agent;
my $agentName = "CiscoSwitchInPerl";
.
.
.
$agent=new RIV::Agent($param, $agentName);
$agent->SendNEToNextPhase($TestNE);
.
.
.
```

Example Usage

The example that illustrates calls to the SendNEToNextPhase and SendNEToDisco methods is divided into the following sections:

- Create a new multi-phased agent
- Setup for discovery phase-dependent processing
- Setup for discovery phase 1 processing
- Setup for discovery phase 2 processing
- Setup for discovery phase 3 processing

Create a new multi-phased agent

```
.
.
.
my $agent;
my $agentName = "CiscoSwitchInPerl";

sub Init{
    my $param=new RIV::Param();
    $agent=new RIV::Agent($param, $agentName);
}
.
.
.
```

The previous code:

- Declares two variables. The *$agent* variable stores the discovery agent application session object identifier returned by the RIV::Agent constructor. The *$agentName* variable stores the name of the agent, which in this example is CiscoSwitchInPerl.
- The call to the RIV::Param constructor returns an object of type RIV::Param to the *$param* variable.
- The call to the RIV::Agent constructor takes two parameters: the RIV::Param object (*$param*) and the name of the agent (*$agentName*). The RIV::Agent constructor returns an agent session object for use in the subsequent call to the SendNEToNextPhase method.

Setup for discovery phase-dependent processing

The following code shows the setup for phase-dependent processing:

```
sub ProcessPhase($){
    my $phaseNumber = shift;

    if($RIV::DebugLevel >= 1)
    {
        print "Phase number is $phaseNumber\n";
    }
}
```

Setup for discovery phase 1 processing

The following code shows the setup for phase 1 processing, including the call to the SendNEToNextPhase method and calls to the SnmpGetNext method. Note that the calls to the SendNEToNextPhase and SnmpGetNext methods are made through the agent session object (*$agent*) returned in the previous call to the RIV::Agent constructor. The SendNEToNextPhase method:

- Marks the network entity (*$TestNE*) as having been processed for phase 1.
- Puts this network entity on the CiscoSwitchInPerl agent queue ready for phase 2 processing.

```
sub ProcessPhase1($){
    my $TestNE = shift;
.
.
.
BuildVlanData($TestNE);
    my $refVlanIfIndex=$agent->SnmpGetNext($TestNE,'vlanIfIndex');
```

```
BuildCardPortToIfIndexData($TestNE);

    my $refLphysAddress=$agent->SnmpGetNext($TestNE,'ifPhysAddress');
.
.
.
$agent->SendNEToNextPhase($TestNE);
}
```

Setup for discovery phase 2 processing

A second call to the SendNEToNextPhase method causes the network entity to be marked as having been processed for phase 2 and to be added to the CiscoSwitchInPerl agent queue ready for phase 2 processing.

```
sub ProcessPhase2($){
    my $TestNE = shift;
.
.
.
$agent->SendNEToNextPhase($TestNE);
}
```

Setup for discovery phase 3 processing

The following code sets up discovery phase 3 processing: Finally, the multi-phased discovery agenta third call to the SendNEToNextPhase method causes the network entity to be marked as having been processed for phase 3 and that it need not be added to the CiscoSwitchInPerl agent queue because there is no additional phase processing required. This multi-phased agent also sends back to the DISCO process the SendNEToNextPhase record set to the value 1 to indicate the final tokenA second parameter to the SendNEToNextPhase method, the value 0 (zero), signifies to the DISCO process that this is the last record token and no additional phase processing is required.

```
sub ProcessPhase3($){
    my $TestNE = shift;
.
.
.
$TestNE->{'m_LastRecord'}=1;
.
.
.
$agent->SendNEToDisco($TestNE,0);
}
```

The CiscoSwitchInPerl discovery agent sends back the data associated with this network entity. Because there is no further processing required for this network entity, the CiscoSwitchInPerl discovery agent:

- Sets the m_LastRecord to the value 1 to indicate the final token and to let the DISCO process know that processing is complete for this network entity.
- Passes the value 0 (zero) as the second parameter in the call to SendNEToDisco. A multi-phased agent receives a single network entity, but it may return to the DISCO process several records (one for each local neighbor entry and one for each remote neighbor entry). The DISCO process determines that the multi-phased discovery agent has finished processing a network entity when the value 0 is specified in the call to SendNEToDisco to indicate the last record token.

Returns

Upon completion, the SendNEToNextPhase method returns no value.

See also

- “RIV::Agent Constructor” on page 188
- “SendNEToDisco” on page 204

SnmpGet

The SnmpGet method retrieves the appropriate SNMP information from the Helper Server.

Method Synopsis

```
SnmpGet($ne, $oid [,$instance, $communitySuffix] )
```

Parameters

$ne
Specifies a reference to the network entity. Typically, this network entity is a RIV::Record object.

$oid
Specifies a MIB variable (for example, *ifIndex*).

$instance
Specifies the instance of the MIB variable. This is an optional parameter.

$communitySuffix
Specifies the suffix to the community string. This is an optional parameter.

Description

The SnmpGet method retrieves the specified SNMP information for the network entity specified in the *$ne* parameter.

Notes

The SnmpGet method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
$varOp = $agent->SnmpGet($NE, 'sysDescr');
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
```

Returns

Upon completion, the SnmpGet method returns a *varop* that contains two key value pairs. The keys are ASN1 and value. The ASN1 value is the index value after the OID corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The ASN1 value obtained using the RIV::SnmpAccess module is the complete OID that needs to be split, whereas the ASN1 value returned by the Helper Server is only the index part.

SnmpGetBulk

The SnmpGetBulk method retrieves SNMP GETBULK information from the Helper Server.

Method Synopsis

```
SnmpGetBulk($ne, $oidList, $nonRepeaters,maxRepeaters [,$communitySuffix] )
```

Parameters

$ne
Specifies a reference to the network entity. Typically, this network entity is a `RIV::Record` object.

$oidList
Specifies a reference to an array of MIB variables. For example:

```
@oids=('sysDescr','sysContact','sysUpTime','ipInReceives',
'ipOutRequests','ipOutDiscards','ipForwDatagrams',
'tcpCurrEstab', 'ifDescr');
$oidList = \@oids;
```

$noRepeaters
Specifies the number of MIB values at the start of the array of MIB variables that return a single value. For example, the `'sysDescr'` MIB variable from the `@oids` array returns a single value.

$maxRepeaters
This parameter is for any MIB variable (for example, `ifIndex`) in the array of MIB variables that returns a table. This parameter specifies the number of values in the table that are to be returned. For example, the value 2 returns only the first two entries. If all the entries are to be returned, *$maxRepeaters* is set to a large number.

$communitySuffix
Specifies the suffix to the community string.

Description

The `SnmpGetBulk` method retrieves SNMP GETBULK information for the network entity specified in the *$ne* parameter.

Notes

The `SnmpGetBulk` method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
@oids=('sysDescr','sysContact','sysUpTime','ipInReceives',
'ipOutRequests','ipOutDiscards','ipForwDatagrams','tcpCurrEstab',
'ifDescr');
($vap) = $agent->SnmpGetBulk($nodeIP, \@oids, 3, 100);
foreach my $varop (@{ $vap})
{
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
}
```

Returns

Upon completion, the `SnmpGetBulk` method returns a reference to a *varop* array. Each *varop* array contains two key value pairs. The keys are `ASN1` and `VALUE`. The `ASN1` value is the index value after the `OID` corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The `ASN1` value obtained using the `RIV::SnmpAccess` module is the complete `OID` that needs to be split, whereas the `ASN1` value returned by the Helper Server is only the index part.

SnmpGetNext

The SnmpGetNext method retrieves the appropriate SNMP information from the Helper Server.

Method Synopsis

```
SnmpGetNext($ne, $oid [,$instance, $communitySuffix] )
```

Parameters

$ne
Specifies a reference to the network entity. Typically, this network entity is a RIV::Record object.

$oid
Specifies a MIB variable (for example, *ifIndex*).

$instance
Specifies the instance of the MIB variable. This is an optional parameter.

$communitySuffix
Specifies the suffix to the community string. This is an optional parameter.

Description

The SnmpGetNext method retrieves the specified SNMP information for the network entity specified in the *$ne* parameter. If *$instance* is defined, the MIB sub-tree starting at that particular instance is retrieved. The *$instance* parameter must be specified as an ASN1 string (for example, "5.3.15").

Notes

The SnmpGetNext method issues the appropriate SNMP request to the Helper Server, which performs the actual work. Thus, the Helper Server (and ncp_ctrl) must be running so that this method can make the appropriate SNMP request.

Example Usage

```
$varOpArray = $agent->SnmpGetNext($NE, 'ifDescr');
foreach my $varop (@{ $varOpArray})
{
print "$varop->{ASN1}", "$varop->{VALUE}", "\n";
}
```

Returns

Upon completion, the SnmpGetNext method returns a reference to a *varop* array. Each *varop* array contains two key value pairs. The keys are ASN1 and VALUE. The ASN1 value is the index value after the OID corresponding to the MIB variable is removed. It is a single number for MIB variables indexed on a single key and a dot notation for MIB variables indexed by multiple keys.

Note: The ASN1 value obtained using the RIV::SnmpAccess module is the complete OID that needs to be split, whereas the ASN1 value returned by the Helper Server is only the index part.

UnLockThreads

The UnLockThreads method releases the lock previously acquired in a call to the LockThreads method.

Method Synopsis

```
UnLockThreads()
```

Parameters

None

Description

The UnLockThreads method releases the lock previously acquired in a call to the LockThreads method.

Example Usage

The following example shows a call to the LockThreads method followed by a call to the UnLockThreads method to release the lock. The example assumes a previous call to the RIV::Agent constructor, which returns a RIV::Agent object (represented by $agent->).

```
$agent->LockThreads();

    #
    # Serialised code goes here
    #

    $agent->UnLockThreads();
```

Returns

Upon completion, the UnLockThreads method does not return any values.

RIV::App module reference

The RIV::App module provides an interface for implementing Network Manager client/server applications within one domain.

The RIV::App module provides two constructors that instantiate a RIV::App object. The constructors are described in reference (man) page format.

Note: The RIV::App module does not provide any methods or functions.

RIV::App module synopsis

The RIV::App module synopsis shows how to make calls to the two constructors that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructors. The reference (man) page for the constructors provides the details.

```
# Load the RIV::App module.
use Riv::App;
#
   # Call the first form of the RIV::App constructor, passing to $domain the
   # name of the Network Manager domain. The constructor returns
   # a RIV::App object to $rivApp.
$rivApp = new RIV::App($domain, $progname, $doHeartBeat);
#
   # Call the second form of the RIV::App constructor, passing as the
   # first parameter a RIV::Param object that was returned
   # in a previous call to the RIV::Param constructor. The constructor
   # returns a RIV::App object to $rivApp.
$rivApp = new RIV::App(RIV::Param, $progname, $doHeartBeat);
```

RIV::App Constructor

The RIV::App constructor creates and initializes a new application session.

Constructor

```
new($domain, $progname [, $doHeartBeat])

new(RIV::Param, $progname [, $doHeartBeat])
```

Parameters

$domain
Specifies a Network Manager domain name.

Note: A default domain name is not supported.

RIV::Param
Specifies a RIV::Param object that was returned in a previous call to the RIV::Param constructor. The RIV::Param object contains a parsed form of the command line arguments.

$progname
Specifies a parameter used when building fault-tolerant server groups. It must contain a string that uniquely identifies the application. By convention, the application name should start with ncp_.

$doHeartBeat
Specifies an optional parameter that indicates whether the application generates a heartbeat signal. If the application generates a heartbeat signal, set this parameter to a nonzero value.

Description

The RIV::App constructors create and initialize a new application session. The constructors differ in that one takes a *$domain* parameter and the other takes a RIV::Param parameter.

Example Usage

```
$app = RIV::App::new("foo", "ncp_test", 1);

#!$NCHOME/bin/ncp_perl
use RIV;
use RIV::App;
my $app = RIV::App::new("MYDOMAIN", "ncp_test");
...
undef $app;
```

Returns

Upon completion, the RIV::App constructors return a RIV::App object that encapsulates the new application session.

See Also

- “RIV::Param Constructor” on page 225

RIV::OQL module reference

The RIV::OQL module provides an interface to communicate with and perform operations on Network Manager internal databases.

The RIV::OQL module provides a constructor that allows you to create a new RIV::OQL session object and within this session object call methods to:

- Connect to a particular service type
- Create new databases and tables
- Query the internal databases
- Insert records into and delete records from the internal databases
- Print and update records that reside in the internal databases

The constructor and methods are described in reference (man) page format.

RIV::OQL module synopsis

The RIV::OQL module synopsis shows how to make calls to the constructor and database operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and database operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Load the RIV::OQL module
use RIV::OQL;

# Call the RIV::OQL constructor, passing to $appSession one of the
# following blessed references:
#
#   + A RIV::Agent object (returned in a previous call to the
#     RIV::Agent constructor)
#   + A RIV::App object (returned in a previous call to the
#     RIV::App constructor)
#
# The $precisionService parameter takes one of the valid Network Manager
# service names, for example, ncp_disco (Disco service).
#
# The calls to the database operation methods are made through a
# reference to the RIV::OQL session object ($oql->) that the RIV::OQL
# constructor returns.
#

$oql = new RIV::OQL($appSession, $precision_Service);

# Call the Send method to send an OQL query to the specified database.
#
$oql->Send($oqlStatement, $returnResults);
#
# Call the CreateDb method to create a database in the Network Manager
# service specified in a previous call to the RIV::OQL constructor.
#
$oql->CreateDB($databaseName);
#
# Call the CreateTable method to create a table in the database.
#
$oql->CreateTable($databaseName, $tableName, \%columnNamesandTypes);
#
# Call the Insert method to insert records into a database table.

$oql->Insert($database, $table, \%record);
#
# Call the Select method to execute a specific OQL command.
oql->Select($database, $table, $columnName);
#
# Call the RIV module's GetResult function to get input data.
my ($type, $data) = $oql->RIV::GetResult(10);
#
# Call the Print method to print the contents of the records obtained
# as a result of this database query.
$oql->Print($data);
#
# Call the Delete method to delete records from the database table.
$oql->Delete($database, $table, $clauseForDeletion);
#
# Call the Update method to update records that currently reside in
# the database.
$oql->Update($database, $table, $setClause, $whereClause);
```

RIV::OQL Constructor

The RIV::OQL constructor creates and initializes a new RIV::OQL object.

Constructor

```
new($rivSession, $rivService)
```

Parameters

$rivSession
Specifies a blessed reference to either a RIV::App or RIV::Agent object.

$rivService
Specifies the name of a service to indicate the internal database to which this OQL session interacts. The following table identifies the available services to which to connect along with their corresponding executable. For example, the service name Disco indicates that the OQL session will interact with the DISCO databases that the ncp_disco executable creates.

Service Name	Executable
Model	ncp_model
Amos	ncp_event
Monitor	ncp_monitor
Class	ncp_class
Store	ncp_store
Ctrl	ncp_ctrl
Helper	ncp_d_helpserv
Disco	ncp_disco

Description

The RIV::OQL constructor creates and initializes a new RIV::OQL session object.

Example Usage

```
$app = new RIV::App();
$oql = new RIV::OQL($app, 'Disco');
```

Returns

Upon completion, the RIV::OQL constructor returns a RIV::OQL session object.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

Close

The `Close` method closes an OQL Client.

Method Synopsis

```
Close()
```

Parameters

None.

Description

The `Close` method closes an OQL Client. If successful, the OQL client is undefined.

Example Usage

```
$oql->Close();
```

Returns

Upon completion, the `Close` method does not return any values.

See Also

- "RIV::OQL Constructor" on page 214

CreateDB

The `CreateDB` method creates a database.

Method Synopsis

```
CreateDB($databaseName)
```

Parameters

$databaseName
Specifies the name of the database to be created.

Description

The `CreateDB` method creates a database with the name *$databaseName* in the specified service. You specified this service in the *$rivService* parameter in a previous call to the `RIV::OQL` constructor.

Example Usage

The following example shows how to create a new database, with the name `foo`, in the Disco service for which an OQL session was created using the `RIV::OQL` constructor.

```
$oql = new RIV::OQL($app, 'Disco');
$oql->CreateDB("foo");
```

Returns

Upon completion, the `CreateDB` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 214“RIV::Agent Constructor” on page 188

CreateTable

The `CreateTable` method creates a database table.

Method Synopsis

```
CreateTable($databaseName, $tableName, \%columnNames)
```

Parameters

$databaseName
Specifies the name of the database in which the table is to be created.

$tableName
Specifies the name of the table to be created.

\%columnNames
Specifies a hash list of the columns in the table. The keys in the hash list refer to the column name and the values are one of the types supported by the OQL syntax.

Description

The `CreateTable` method creates a database table with the name specified in the *$tableName* parameter in the database specified in the *$databaseName* parameter.

You created this database in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.

Example Usage

The following example shows how to create:

- A new database, with the name `foo`, in the Disco service for which a `RIV::OQL` session object was created in a call to the `RIV::OQL` constructor.
- Column names (`m_IpAddress` and `m_Name`) and associated values to appear in the table.
- A table called `bar`.

```
$oql = new RIV::OQL($app, 'Disco');
$oql->CreateDB("foo");
%columnNames = ("m_IpAddress"=> "text", "m_Name"=> "text");
$oql->CreateTable("foo", "bar", \%columnNames);
```

Returns

Upon completion, the `CreateTable` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 214
- “CreateDB” on page 215

Delete

The `Delete` method deletes records from a database table.

Method Synopsis

```
Delete($databaseName, $tableName, $clause)
```

Parameters

$databaseName
Specifies the name of the database from which records are to be deleted.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) from which records are to be deleted.

$clause
Specifies any valid OQL comparative statement used as a condition for deleting records. If a record matches *$clause*, the `Delete` method will delete it.

Description

The `Delete` method deletes records from the table specified in the *$tableName* parameter that resides in the database specified in *$databaseName* parameter and that satisfy the criteria defined by the OQL comparative statement specified in the *$clause* parameter.

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example shows how to delete records from:

- A database called `ncimCache`.
- A table called `entityData`.

The records to be deleted are those with a description equal to the value `foo`.

```
$oql->Delete('ncimCache', 'entityData', "description = 'foo'");
```

Returns

Upon completion, the `Delete` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 214
- “CreateDB” on page 215
- “CreateTable” on page 216

Insert

The `Insert` method inserts records into a database table.

Method Synopsis

```
Insert($databaseName, $tableName, \%record)
```

Parameters

$databaseName
Specifies the name of the database in which the record is to be inserted.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) in which the record is to be inserted.

\%record
Specifies a hash list that defines the record to be inserted.

Description

The `Insert` method creates an OQL statement that inserts the record defined by the hash list specified in the *\%record* parameter into the database table specified in the *$tableName* parameter that resides in the database specified in the *$databaseName* parameter.

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example shows how to insert the record specified in the *\%record* parameter in:

- A database called `finders`.
- A table called `despatch`.

Note:

The service used to create the `RIV::OQL` object (`$oql->`) in a previous call to the `RIV::OQL` constructor has to be `Disco`.

```
%record = ( m_Creator => 'PerlDetails',
m_Name => 'foo',
m_IpAddress => '123.1.2.3',);
$oql->Insert('finders', 'despatch', \%record);
```

Returns

Upon completion, the `Insert` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 214
- “CreateDB” on page 215

- “CreateTable” on page 216

Print

The `Print` method prints records obtained as a result of a database query.

Method Synopsis

```
Print($data)
```

Parameters

$data
Specifies a reference to an array of hash lists that represent the records obtained from the SELECT statement.

Description

The `Print` method prints the records obtained as a result of a query.

Example Usage

The following example shows how to:

- Use the `Select` method to execute a SELECT statement.
- Use the `RIV::GetResult` method to specify the number of seconds to wait (in the example, 10 seconds) for input before returning.
- Print the data specified in *$data*.

```
$oql->Select('class','activeClasses', 'ALL');
my ($type, $data) = $oql->RIV::GetResult(10);
$oql->Print($data);
```

Returns

Upon completion, the `Print` method does not return any records.

See Also

- “RIV::GetResult” on page 178

Query

The `Query` method runs a query.

Method Synopsis

```
Query($queryString)
```

Parameters

$queryString
Is a string that contains the details of the query.

Description

The `Query` method is used to run a query by using the OQL client subject. Running a query clears any results from a previous query.

Example Usage

```
$oql->Query("select * from disco.status;");
```

Returns

Upon completion, the `Query` method does not return any values.

See Also

- “RIV::OQL Constructor” on page 214
- “QueryGetResult” on page 220
- “QueryGetResults” on page 221

QueryGetResult

The `QueryGetResult` method gets a single result from the `Query` command.

Method Synopsis

```
QueryGetResult()
```

Parameters

None.

Description

The `QueryGetResult` method gets a single result from the `Query` method. The `QueryGetResult` method is commonly used when you don't want to process every result of the query into a Perl reference, to prevent use of much memory. The `QueryGetResult` method can also be used where only a single result is expected. Each time a result is retrieved it is removed from the result stack within the OQL client.

Example Usage

```
my $query = "select * from disco.status;";
$oql->Query($query);

my $data = $oql->QueryGetResult();
if($data)
{
print "Got result.\n";
print Dumper($data);
}
else
{
print "Query $query returned nothing.\n";
}
```

Returns

Upon completion, the `QueryGetResult` method returns a hash reference that contains the details of the returned record.

See Also

- “RIV::OQL Constructor” on page 214
- “Query” on page 219
- “QueryGetResults” on page 221

QueryGetResults

The QueryGetResults method gets multiple results from the Query command.

Method Synopsis

```
QueryGetResults()
```

Parameters

None.

Description

The QueryGetResults method gets all the results from a previously run Query method. This QueryGetResults method is commonly used when you expect the query to return a limited number of records. As each record is turned into a hash reference, it can use much memory. When the results are retrieved, the results are no longer available within the OQL client.

Example Usage

```
my $query = "select * from disco.agents;";
$oql->Query($query);

my $queryData = $oql->QueryGetResults();
if($queryData)
{
$recCount = scalar(@$queryData);

print "query $query returned $recCount results.\n";

foreach my $dat (@$queryData)
{
print "Got result.\n";
print Dumper($dat);
}
}
else
{
print "Query $query returned nothing.\n";
}
```

Returns

Upon completion, the QueryGetResults method returns a reference to a hash list, which contains the results of the query.

See Also

“RIV::OQL Constructor” on page 214
“Query” on page 219
“QueryGetResult” on page 220

Select

The Select method executes a specific OQL statement.

Method Synopsis

```
Select($databaseName, $tableName, $columnName)
```

Parameters

$databaseName
Specifies the name of the database in which the OQL statement is to be executed.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) in which the OQL statement is to be executed.

$columnName
Specifies the name of the column for which the results are to be returned. If all entries are to be returned, set the *$columnName* parameter to ALL.

Description

The Select method executes the following OQL statement:

```
select $columnName from $dbName.$tableName;
```

When the *$columnName* parameter is set to ALL, the Select method executes the following OQL statement:

```
select * from $dbName.$tableName;
```

You created this database and database table in previous calls to the:

- RIV::OQL constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this RIV::OQL session object interacts.
- CreateDB method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the RIV::OQL constructor.
- CreateTable method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the CreateDB method.

Example Usage

```
$oql->Select('class', 'activeClasses', 'ALL');
```

The results are obtained by using the RIV::GetResult method. For example:

```
my ($type, $data) = $oql->RIV::GetResult(10);
```

In the previous example:

- The *$type* parameter specifies the tag OQLQuery.
- The *$data* parameter specifies a reference to an array of hash lists that represents the records obtained from the OQL database query.
- All records are received from the database table called activeClasses that resides in the database called class. In this case, the service to which this OQL session is connected must be Class.

Returns

The results are obtained by using the RIV::GetResult method.

See Also

- “RIV::GetResult” on page 178
- “RIV::OQL Constructor” on page 214
- “CreateDB” on page 215
- “CreateTable” on page 216

Send

The Send method provides a way to communicate with the databases.

Method Synopsis

```
Send($statement, $returnResults)
```

Parameters

$statement
Specifies any valid OQL statement.

$returnResults
Specifies whether to return results. This parameter takes one of the following values:

- 1 – Specify the value 1 for database queries (for example, OQL statements such as `select` and `show`) that return results.
- 0 – Specify the value 0 (zero) for database queries (for example, OQL statements such as `insert`, `update`, and `delete`) that do not return results.

Description

The Send method provides a way to communicate with the databases. The *$statement* parameter specifies any valid OQL statement that the Send method executes. The *$returnResults* parameter indicates whether you are interested in the results of the OQL statement. For example, when an OQL `select` statement is executed and you are interested in the results, the *$returnResults* parameter must be set to the value 1. The `RIV::GetResult` method is used to receive the results.

Example Usage

```
$statement = "select * from ncimCache.entityData;";
$oql->Send($statement, 1);
my ($type, $data) = $oql->RIV::GetResult(10);
```

Returns

Upon completion, the Send method returns the results of the OQL statement. If you set *$returnResults* to 0 (zero), the Send method does not return any records.

See Also

- “RIV::GetResult” on page 178
- “Select” on page 221

Update

The `Update` method updates records that currently reside in the database.

Method Synopsis

```
Update($databaseName, $tableName, $setClause, $whereClause)
```

Parameters

$databaseName
Specifies the name of the database in which the record is to be updated.

$tableName
Specifies the name of the table in the specified database (*$databaseName*) in which the record is to be updated.

$setClause
Specifies the clause that defines `set the variable to`.

$whereClause
Specifies the clause that defines `where the variable is`.

Description

The `Update` method updates records that already reside in the database and database table specified in the *$databaseName* and *$tableName* parameters, respectively. Calling the `Update` method is equivalent to executing the following OQL statement:

```
UPDATE $databaseName.$tableName SET $setClause WHERE $whereClause;
```

You created this database and database table in previous calls to the:

- `RIV::OQL` constructor — You specified the name of a service (in the *$rivService* parameter) to indicate the internal database to which this `RIV::OQL` session object interacts.
- `CreateDB` method — You specified the name of the database (in the *$databaseName* parameter) to be created in the service specified in the call to the `RIV::OQL` constructor.
- `CreateTable` method — You specified the name of the database table (in the *$tableName* parameter) to be created in the database specified in the call to the `CreateDB` method.

Example Usage

The following example does the following:

- Updates specific records in the table called `entityData` that resides in the database called `ncimCache`.
- The specific records updated are those that have `EntityName foo` with a `description foo`.

```
$oql->Update('ncimCache', 'entityData', "EntityName='foo'",
"description='foo'");
```

Returns

Upon completion, the `Update` method does not return any records.

See Also

- “RIV::OQL Constructor” on page 214
- “CreateDB” on page 215
- “CreateTable” on page 216

RIV::Param module reference

The `RIV::Param` module provides an interface for parsing standard and Network Manager application-specific command line arguments.

The `RIV::Param` module provides a constructor that creates a new `RIV::Param` object that you use to call methods that perform the following tasks:

- Obtain the name of a command
- Obtain the name of a domain
- Print a brief usage explanation to standard output

The constructor and methods are described in reference (man) page format.

RIV::Param module synopsis

The `RIV::Param` module synopsis shows how to make calls to the constructor and parameter operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and parameter operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Load the RIV::Param module
use RIV::Param;
#
# These are the RIV::Param module constants used to specify
# whether a command line parameter takes no arguments or a
# single argument and whether it is mandatory.
RivParamNoArg, RivParamSingleArg;
RivParamMandatory, RivParamListArg;
#
# Call the RIV::Param constructor. The RIV::Param constructor
# returns to $param a new RIV::Param object.
#

$param = RIV::Param::new(\%paramHash, \$usageStringsRef, \$helpMessage, $dieOnUnknownArgs);
#
# Use the RIV::Param object ($param->) to invoke the methods
# that the RIV::Param module provides.

# Call the Usage method to print a brief usage explanation to
# standard output.
$param->Usage($errorCode);
#
# Call the DomainName method to obtain the name of the domain.
$domainName = $param->DomainName();
#
# Call the CommandName method to obtain the name of the command.
$commandName = $param->CommandName();
```

RIV::Param Constructor

The `RIV::Param` constructor creates and initializes a new `RIV::Param` object.

Constructor

```
$param = RIV::Param::new([\%paramHash,\@usageStrings, \$helpMessage])
```

Parameters

\%paramHash

Specifies a reference to a hash used to specify application-specific command line arguments. Each hash key (index) represents a command line switch and its associated hash value is an array with the following elements:

- element 0 — Is a `flags` element that specifies a bitwise OR that indicates:
  - Whether the command line switch takes an argument.
  - Whether the switch is mandatory

  The `flags` element makes use of the package constants described in “Package Constants” on page 226.
- element 1 — Is a scalar variable reference or `undef` value. You initialize the scalar variable reference with the appropriate value (a parameter from the command line or 1).

The *\%paramHash* parameter is optional.

\$usageStringsRef
This is an array reference that contains an element for each of the non-standard command line argument scenarios. If the application takes only standard arguments, this constructor argument should be set to `undef`.

By default, the usage message has the standard arguments appended to it (`-domain`, `-debug`, `-latency`, `-messageLevel`, `-help`). If the application does not accept all of these, or you do not want to print them, then include an element in the array with the value `nostdargs`. This element will not be printed, but will prevent the usage message from listing the standard arguments.

\$helpMessage
Specifies a string reference that contains explanatory information that is written to standard output, in addition to standard help information, when the `-help` command line argument is specified.

The *\$helpMessage* parameter is optional.

$dieOnUnknownArgs
This is an optional scalar. If present and true, and if the command line (@ARGV) contains any parameters that are not in `\%paramHashRef` or the default set of accepted parameters, this causes the constructor to fail (return `undef`). If this parameter is not present or is false, the constructor silently ignores any unrecognized parameters, because some callers prefer to do a part of their own command-line processing.

Package Constants

The `RIV::Param` constructor's *\%paramHash* parameter makes use of the following package constants:

- `RivParamNoArg` — Specifies that the command line parameter takes no arguments.
- `RivParamSingleArg` — Specifies that the command line parameter takes one argument.
- `RivParamMandatory` — Specifies that the command line parameter is mandatory, and that it is a fatal error for the parameter to be missing.
- `RivParamListArg` — Specifies that the command-line parameters accepts a list of zero or more values.

These constants are bit masks that can be ORed together where appropriate.

For example, `RivParamMandatory | RivParamSingleArg` specifies a mandatory parameter that requires a single argument.

Description

The `RIV::Param` constructor creates and initializes a new `RIV::Param` object from the application-specific command line arguments specified in the *\%paramHash* parameter. Each new `RIV::Param` object also encapsulates the supported standard command line arguments. Thus, Network Manager client/server and Agent applications can make use of these standard command line arguments in addition to the application-specific command line arguments.

If you call the `RIV::Param` constructor without specifying any of the optional parameters, the new `RIV::Param` object provides access to the standard command line arguments.

Example Usage No Parameters

The following code shows a call to the `RIV::Param` constructor without specifying any of the optional parameters. The newly created `RIV::Param` object is then passed to the `RIV::Agent` constructor, which returns a `RIV::Agent` object that provides a discovery agent application session. In this example, the discovery agent called `PerlDetails` (the name specified in the second parameter of the `RIV::Agent` constructor) can make use of the standard command line arguments.

```
.
.
.
sub Init{
```

```
    my $param=RIV::Param::new();
    $agent=RIV::Agent::new($param,"PerlDetails");
}
.
.
.
```

Example Usage Three Parameters

The example described in this section shows a call to the `RIV::Param` constructor that specifies the three optional parameters.

The following code defines a reference to a hash called `%CmdLineArgs` used to specify application-specific command line arguments. The `%CmdLineArgs` hash is passed as the first parameter to the `RIV::Param` constructor:

```
.
.
.
my $subject;
my $process = 'Model';
my $messageClass = 'NOTIFY';
my $verbose;
my %CmdLineArgs = (
    "-subject"  => [ RivParamSingleArg , \$subject ],
    "-process"  => [ RivParamSingleArg , \$process ],
    "-messageClass"  => [ RivParamSingleArg , \$messageClass ],
    "-verbose"  => [ RivParamNoArg,     \$verbose ]
);
.
.
.
```

The following list provides brief descriptions of the application-specific command line arguments defined in the `%CmdLineArgs` hash.

- `-subject` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$subject`. It is expected that a user would supply a specific subject on the command line.
- `-process` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$process`. It is expected that a user would supply the specific process that is of interest (for example, `Class`, `Config`, `Event`, and so forth) on the command line. The default process is `Model`.
- `-messageClass` — Specifies a command line argument that takes one argument (as indicated by the `RivParamSingleArg` package constant). This command line argument also specifies a reference to a scalar value, `\$messageClass`. It is expected that a user would supply the class of messages that are of interest (for example, `QUERY`, `STATUS`, and so forth) on the command line. The default message class is `NOTIFY`.
- `-verbose` — Specifies a command line argument that takes no arguments (as indicated by the `RivParamNoArg` package constant). This command line argument also specifies a reference to a scalar value, `\$verbose`. It is expected that a user would specify this command line argument to explicitly print out details of nested fields.

The following code defines a usage string called `@Usage` that provides information on how to use the command line for this application. The `@Usage` array is passed as the second parameter to the `RIV::Param` constructor:

```
.
.
.
my @Usage = (
    "[-subject <subject> -process [Model|Disco|Ctrl|...]
     -messageClass [NOTIFY|QUERY|...] "
);
.
```

```
.
.
```

The following code defines a string reference called $helpData that contains explanatory information about the application-specific command line arguments and other pertinent information. The explanatory information also includes descriptions of the standard command line arguments (-domain, -debug, and -help). The $helpData string reference is passed as the third parameter to the RIV::Param constructor:

```
my $helpData = "\n
The ITListener perl script is intended to listen on the supplied subject
and print out the messages received.

The arguments are
-domain <domain> = Name of the domain to retrieve data from
-debug [0-4] = Required debug level
-help = This information
-verbose = Explicitly print out details of nested fields
-subject = The specific subject to listen to ( this will not include the domain )
-process = The process to listen to ( e.g. Model, Class, Event, Config, Ctrl ,
           Disco , PingFinder )
-messageClass = The class of messages of interest. Not all processes support
all classes. The common ones of interest are NOTIFY, QUERY, STATUS

The most common arguments to use are
-process Model -messageClass NOTIFY : ( default ) - Listen for the models
 updates on topology changes (old style)
-process Model -messageClass TOPOLOGY : ( default ) - Listen for the models
 updates on topology changes (new style)
-process Disco -messageClass STATUS : - Listen to disco broadcasts on the
 current state of the discovery.
-process DNCIM2NCIM -messageClase NOTIFY : - Listen to Disco to Model
 DNCIM2NCIM updates.
-process ITNMSTATUS -messageClass NOTIFY : - Listen to ITNM status events.

The process is capable of listening on any subject on the message broker bus
but will not decode the output beyond printing out the contents of the message.

The syntax for message broker subjects is

    /<subject>/<sub-subject>/<sub-sub-subject>/....

All ITNM IP subjects begin \'ITNM/\' and have the domain appended so the
model notify subject for domain TESTDOMAIN is

    /ITNM/MODEL/NOTIFY/TESTDOMAIN



\n";
```

The following code shows the call to the RIV::Param constructor using the three previously defined parameters: \%CmdLineArgs, \@Usage, and \$helpData. Note the call to the die function to exit the script if the RIV::Param constructor fails to create a new RIV::Param object.

```
.
.
.
my $param = RIV::Param::new(\%CmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
```

Returns

Upon completion, the RIV::Param constructor returns a new RIV::Param object.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

- “Usage” on page 230
- “RIV::Param module overview” on page 126

CommandName

The CommandName method returns the name of the specified command.

Method Synopsis

```
RIV::Param::CommandName()
```

Parameters

None

Description

The CommandName method returns the name of the command specified on the command line.

Use the RIV::Param object returned in a previous call to the RIV::Param constructor to invoke the CommandName method. For example: $param->CommandName.

Example Usage

The following code shows a call to the CommandName method. Note that the CommandName method is invoked through the newly created RIV::Param object returned to the *$param* variable.

```
.
.
.
my $param = RIV::Param::new(\%cmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
  my $command = $param->CommandName();
.
.
.
```

Returns

Upon completion, the CommandName method returns the name of the command specified on the command line.

See Also

- “RIV::Param Constructor” on page 225

DomainName

The DomainName method returns the name of the specified domain.

Method Synopsis

```
RIV::Param::DomainName()
```

Parameters

None

Description

The DomainName method returns the name of the domain specified on the command line for the -domain standard command line argument.

Use the RIV::Param object returned in a previous call to the RIV::Param constructor to invoke the DomainName method. For example: $param->DomainName.

Example Usage

The following code shows a call to the DomainName method. Note that the DomainName method is invoked through the newly created RIV::Param object returned to the *$param* variable.

```
.
.
.
my $param = RIV::Param::new(\%cmdLineArgs, \@Usage, \$helpData);
die "Can't create RIV::Param" unless (defined $param);
.
.
.
die "ncp_disco must be running under domain ",
        $param->DomainName(),
        " - unable to query the disco.config table"
        unless $dbData;

    print "...disco is running under domain ", $param->DomainName(), "\n" if $debug;
.
.
.
```

Returns

Upon completion, the DomainName method returns the name of the domain specified on the command line for the -domain standard command line argument.

See Also

- “RIV::Param Constructor” on page 225

Usage

The Usage method writes a brief usage explanation to standard output.

Method Synopsis

```
RIV::Param::Usage($errorCode)
```

Parameters

$errorCode
Specifies either a status or the undef value. The status gets written to standard output.

Description

The Usage method writes a brief usage explanation to standard output and then exits with the status specified in the *$errorCode* parameter, if defined. If you specified the undef value in the *$errorCode* parameter, the Usage method returns to the caller.

Use the RIV::Param object returned in a previous call to the RIV::Param constructor to invoke the Usage method. For example: $param->Usage.

Example Usage

The following code shows a call to the Usage method. In this example, an error code of 1 is passed. Note that the Usage method is invoked through the newly created RIV::Param object returned to the *$param* variable.

```
my @_Usage = (                # usage string suffixes
    "<node> [ async ]"
);

#
# Read and parse the command line, standard args are hidden
#
my $param = RIV::Param::new({
        "-v"        => [ $RIV::Param::NoArg, \$Verbose ],
    }, \@_Usage);
die "RIV::Param::new failed" unless defined $param;

my $node        = shift @ARGV;
my $what        = shift @ARGV;
$what = "" unless defined $what;

$param->Usage(1)
    unless (defined $node && $node ne "");
```

Returns

Upon completion, the Usage method writes a brief usage message to standard output and the status specified in the *$errorCode* parameter and simply exits. If the *$errorCode* parameter is set to the undef value, the Usage method returns to the caller.

See Also

- "RIV::Param Constructor" on page 225

RIV::Record module reference

The RIV::Record module provides a data structure to store the network entity.

The RIV::Record module provides a constructor that creates and initializes a RIV::Record data structure. This module also provides methods to perform the following operations:

- Add local neighbors
- Add remote neighbors
- Get local neighbors
- Get remote neighbors
- Print the current records

The constructor and methods are described in reference (man) page format.

RIV::Record module synopsis

The RIV::Record synopsis shows how to make calls to the constructor and local and remote neighbors operation methods that this module provides.

```
use RIV::Agent;
use RIV::Record;
my($tag, $data) = $agent->RIV::GetResult(-1);
if($tag eq 'NE'){
foreach $key (@$data){
$NE = RIV::Record::new($key);
}
}
$NE->AddLocalNeighbour($refLocalNeighbour);
$NE->AddRemoteNeighbour($refLocalNeighbour, $refRemoteNeighbour);
```

```
$arrayVarOps = $agent->SnmpGetNext($NE, $mibVariable);
$NE->AddLocalNeighbourTag($tagName, $arrayVarOps);
$NE->AddRemoteNeighbourTag($reflocalNeighbour, $tagName, $arrayVarOps);
@localNeighbours = $NE->GetLocalNeighbours();
@remoteNeighbours = $NE->GetRemoteNeighbours($refLocalNeighbour);
$NE->Print();
```

RIV::Record Constructor

The RIV::Record constructor creates and initializes a new RIV::Record object.

Constructor

```
new($refNE)
```

Parameters

$refNE

Specifies a reference to a hash list. The hash list is the mechanism used to store network entity records retrieved from the discovery engine, DISCO.

Description

The RIV::Record constructor creates and initializes a new RIV::Record object. This object stores network entity records retrieved from DISCO.

Example Usage

The following code fragment illustrates a typical loop for receiving records from DISCO:

```
while (1){
my($tag, $data) = $agent->RIV::GetResult(-1);
# Get the network entities
print "TAG :", $tag, "\n";
if($tag eq 'NE'){
foreach $key (@$data){
$ne = new RIV::Record($key);
}
}
}
```

Returns

Upon completion, the RIV::Record constructor returns a RIV::Record object.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::GetResult” on page 178

AddLocalNeighbour

The AddLocalNeighbour method adds a local neighbor.

Method Synopsis

```
AddLocalNeighbour($refNbr)
```

Parameters

$refNbr
Specifies a reference to a hash list that defines the local neighbor using a set of key value pairs (varBinds).

Description

The AddLocalNeighbour method adds a local neighbor whose hash list reference is *$refNbr*.

Example Usage

```
$localNbr{'m_IpAddress'} = '1.2.3.4';
$localNbr{'m_IfIndex'} = 2;
$NE->AddLocalNeighbour(\%localNbr);
```

Returns

Upon completion, the AddLocalNeighbour method does not return any records.

AddLocalNeighbourTag

The AddLocalNeighbourTag method adds a tag (varBind) to a local neighbor.

Method Synopsis

```
AddLocalNeighbourTag($tag, $refVarOp)
```

Parameters

$tag
Specifies the key value for the varBind.

$refVarOp
Specifies a reference to an array of varops.

Description

The AddLocalNeighbourTag method adds to local neighbors a varBind whose key is defined by the *$tag* parameter and the value defined by the *$refVarOp* parameter (a reference to an array of varops). The key and value are added sequentially, that is, the values in the @$refVarOp array are assumed to be in the same order as the local neighbor array. If local neighbors do not exist, then AddLocalNeighbourTag creates them.

Example Usage

```
$refLifindex=$agent->SnmpGetNext($TestNE, 'ipAdEntIfIndex');
$TestNE->AddLocalNeighbourTag("m_IfIndex", $refLifIndex);
```

Returns

Upon completion, the AddLocalNeighbourTag method does not return any records.

See Also

- "RIV::Agent Constructor" on page 188
- "SnmpGetNext" on page 210

AddRemoteNeighbour

The AddRemoteNeighbour method adds a remote neighbor.

Method Synopsis

```
AddRemoteNeighbour($refLocalNbr, $refRemoteNbr)
```

Parameters

$refLocalNbr
Specifies a reference to the hash list that defines a local neighbor and to which list the remote neighbor is to be added.

$refRemoteNbr
Specifies a reference to a hash list that defines the remote neighbor using a set of key value pairs (varBinds).

Description

The AddRemoteNeighbour method adds a remote neighbor whose hash list reference is *$refRemoteNbr* to the local neighbor whose hash list reference is *$refLocalNbr*.

Example Usage

```
$remoteNbr{'m_IpAddress'} = '1.2.5.6';
$NE->AddRemoteNeighbour($localNbr, \%remoteNbr);
```

Returns

Upon completion, the AddRemoteNeighbour method does not return any records.

AddRemoteNeighbourTag

The AddRemoteNeighbourTag method adds a tag (varBind) to a remote neighbor.

Method Synopsis

```
AddRemoteNeighbourTag($refLocalNbr, $tag, $refVarOp)
```

Parameters

$refLocalNbr
Specifies a reference to the local neighbor to which the remote neighbors are to be added.

$tag
Specifies the key value for the varBind.

$refVarOp
Specifies a reference to an array of varops.

Description

The AddRemoteNeighbourTag method adds to remote neighbors a varBind whose key is defined by the *$tag* parameter and the value defined by the *$refVarOp* parameter (a reference to an array of varops). The key and value are added sequentially, that is, the values in the @$refVarOp array are assumed to be in the same order as the remote neighbor array. If remote neighbors do not exist, then AddRemoteNeighbourTag creates them.

The *$tag* parameter specifies a reference to the local neighbor to which the remote neighbor currently resides or will be added (if it does not currently exist).

Example Usage

```
$refRifIndex = $agent->SnmpGetNext($TestNE,...);
$TestNE->AddRemoteNeighbourTag($refLocal, "m_IfIndex", $refRifIndex);
```

Returns

Upon completion, the AddRemoteNeighbourTag method does not return any records.

See Also

- "RIV::Agent Constructor" on page 188
- "SnmpGetNext" on page 210

GetLocalNeighbours

The GetLocalNeighbours method returns an array of local neighbors.

Method Synopsis

```
GetLocalNeighbours()
```

Parameters

None

Description

The GetLocalNeighbours method returns an array of local neighbors.

Example Usage

```
@localNeighbours = $NE->GetLocalNeighbours();
```

Returns

Upon completion, the GetLocalNeighbours method returns an array of local neighbors (as a reference to a hash list).

GetRemoteNeighbours

The GetRemoteNeighbours method returns an array of remote neighbors.

Method Synopsis

```
GetRemoteNeighbours($refLocalNeighbour)
```

Parameters

$refLocalNeighbour
Specifies a reference to the hash list that defines a local neighbor and for which list the remote neighbor is to be returned.

Description

The GetRemoteNeighbours method returns an array of remote neighbors associated with the specified local neighbor. The local neighbor is specified in the hash list passed to the *$refLocalNeighbour* parameter.

Example Usage

```
@remoteNeighbours = $NE->GetRemoteNeighbours($refLocalNeighbour);
```

Returns

Upon completion, the GetRemoteNeighbours method returns an array of remote neighbors (as a reference to a hash list).

Print

The Print method prints the current record.

Method Synopsis

```
Print()
```

Parameters

None

Description

The Print method prints the current record.

Example Usage

```
$NE->Print();
```

Returns

Upon completion, the Print method does not return any records.

RIV::RecordCache module reference

The RIV::RecordCache module provides an interface to access a record cache file.

The RIV::RecordCache module provides a constructor that creates and initializes a RIV::RecordCache file object. After creating this object, you can call methods that:

- Create or open an existing RIV::RecordCache file object
- Add a record to this RIV::RecordCache file object and obtain the key under which this record was added
- Retrieve all the records that reside in this RIV::RecordCache file object
- Retrieve a specific record from this RIV::RecordCache file object using the record's associated key

The constructor and methods are described in reference (man) page format.

RIV::RecordCache module synopsis

The RIV::RecordCache module synopsis shows how to make calls to the constructor and record cache operation methods that this module provides.

```
use RIV::RecordCache;
  $recordCache = new RIV::RecordCache($rivSession, $cacheName [ ,$cacheLocation ] );
  my $recKey = $recordCache->CacheRecord($record);
  $recordCache->GetRecords();
  $recordCache->GetRecord($recKey);
```

RIV::RecordCache Constructor

The RIV::RecordCache constructor creates and initializes a new RIV::RecordCache file object.

Constructor

```
new($rivSession, $cacheName [,$cacheLocation])
```

Parameters

$rivSession
Specifies a blessed reference to either a RIV::App or RIV::Agent object. More specifically, this is a RIV::App or RIV::Agent application object returned in a previous call to the RIV::App or RIV::Agent constructor.

$cacheName
Specifies the name of the RIV::RecordCache file object to be created or read from.

$cacheLocation
Specifies the path to the RIV::RecordCache file object.

This parameter is optional. If you do not pass a value to this parameter, the path to the RIV::RecordCache file object is assumed to be the $NCHOME/var/precision directory.

Description

The RIV::RecordCache constructor creates and initializes a new RIV::RecordCache file object with the name as specified in the *$cacheName* parameter and the location as specified in the *$cacheLocation* parameter.

Example Usage

The following code fragment illustrates a typical call to the RIV::RecordCache constructor:

```
$app = RIV::App::new();
    $cache = RIV::RecordCache::new($app, "Disco.Cache.Details.returns.MYDOMAIN",
                                   "/opt/netcool/var/precision/");
}
```

Returns

Upon completion, the RIV::RecordCache constructor returns a RIV::RecordCache file object. This is the object upon which you can perform add and retrieve record operations.

See Also

- "RIV module reference" on page 165
- "RIV::Agent Constructor" on page 188
- "RIV::App Constructor" on page 212

CacheRecord

The `CacheRecord` method attempts to add the specified record to the specified cache.

Method Synopsis

```
CacheRecord($record)
```

Parameters

$record
Specifies the record that is to be added to the cache. This record is expressed as a hash.

Description

The `CacheRecord` method adds the record specified in the *$record* parameter to the specified cache. You specified the name of the cache in a previous call to the `RIV::RecordCache` constructor.

Example Usage

The following example illustrates a typical call to the `CacheRecord` method, where the method caches the record (hash) called *$myRec*:

```
$cache->CacheRecord($myRec);
```

Returns

Upon completion, the `CacheRecord` method returns:

- The value -1 to indicate that the attempt to add the record to the cache was unsuccessful. The method displays an appropriate error message requesting that you check to ensure that the cache is valid.
- The key that the record was added under if the attempt to add the record to the cache was successful.

See Also

- “RIV::RecordCache Constructor” on page 237

GetRecord

The `GetRecord` method retrieves from the cache a record associated with the specified key.

Method Synopsis

```
GetRecord($recordKey)
```

Parameters

$recordKey
Specifies the key associated with the record to be retrieved from the cache. This key was returned in a previous call to the `CacheRecord` method after it successfully inserted the record into the cache.

Description

The `GetRecord` method retrieves from the cache a record associated with the key specified in the *$recordKey* parameter. You specified the name of the cache in a previous call to the `RIV::RecordCache` constructor.

Example Usage

The following example illustrates a typical call to the `GetRecord` method, where the method returns to *$record* a hash from a previously specified cache that contains several records:

```
my $record = $cache->GetRecord();
```

Returns

Upon completion, the `GetRecord` method returns:

- *%record* — Specifies a hash that represents one of the records residing in the cache.

See Also

- “RIV::RecordCache Constructor” on page 237
- “CacheRecord” on page 238
- “GetRecords” on page 239

GetRecords

The `GetRecords` method retrieves from the cache a list of all the records currently residing in it.

Method Synopsis

```
GetRecords()
```

Parameters

None

Description

The `GetRecords` method retrieves from the cache a list of all the records currently residing in it. Each record is returned as a hash within a list.

Example Usage

The following example illustrates a typical call to the `GetRecords` method, where the method returns to *recordList* an array of hashes from a previously specified cache that contains several records:

```
my @recordList = $cache->GetRecords();
```

Returns

Upon completion, the `GetRecords` method returns:

- *$recordList*— Specifies an array of hashes, where each hash represents one of the records in the cache.

See Also

- “RIV::RecordCache Constructor” on page 237

- "CacheRecord" on page 238
- "GetRecord" on page 238

RIV::SnmpAccess module reference

The RIV::SnmpAccess module provides an interface to perform SNMP-related operations on Network Manager MIB trees.

The RIV::SnmpAccess module provides a constructor that allows you to create and initialize a new RIV::SnmpAccess session object. After obtaining this session object, you can call the synchronous or asynchronous versions of the SnmpGet-related methods to perform the following operations:

- SNMP get
- SNMP get-next
- SNMP get-bulk

The RIV::SnmpAccess module also provides several utility methods that allow you to operate on ANS.1 (Abstract Syntax Notation One) values and the MIB tree.

Note: Discovery agents implemented with this version of the Perl API should use the SNMP methods that the RIV::Agent module provides to obtain SNMP information from a network device.

The constructor and methods are described in reference (man) page format.

RIV::SnmpAccess module synopsis

The RIV::SnmpAccess module synopsis shows how to make calls to the constructor and SNMP operation methods that this module provides.

Synopsis

```
use RIV::SnmpAccess;

$RIV::SnmpAccess::MaxAsyncConcurrent;

$snmp = new RIV::SnmpAccess($RivSession);

\%varop = $snmp->SnmpGet($host, $addOn, $oid
[, $instance [, $splitOutput]]);

$ok = $snmp->AsyncSnmpGet($tag, $host, $addOn, $oid
[, $instance [, $splitOutput]]);

\@varops = $snmp->SnmpGetNext($host, $addOn, $oid
[, $instance [, $splitOutput]]);

$ok = $snmp->AsyncSnmpGetNext($tag, $host, $addOn, $oid
[, $instance [, $splitOutput]]);

\@varops = $snmp->SnmpGetBulk($host, $addOn, $oidBindList,
 $nonRepeaters, $maxRepetitions [, $instance [, $splitOutput]]);

$ok = $snmp->AsyncSnmpGetBulk($tag, $host, $addOn, $oidBindList, $nonRepeaters,
$maxRepetitions [, $instance [, $splitOutput]]);

(\@varops, \%status) = $snmp->SnmpWalk($host, $addOn, $oid,
$ignoreInstanceFilters);

where:

$asn1 = $varop{ASN1};
$value = $varop{VALUE};

foreach my $vp (@varops) {
$asn1 = $vp->{ASN1};
$value = $vp->{VALUE};
...
}

($baseOid, $indexOid, $baseOidName) = $snmp->SplitOidAndIndex($fullASN1);
```

```
($fullASN1);

$asn1 = $snmp->OidToASN1($oid);
```

RIV::SnmpAccess Constructor

The RIV::SnmpAccess constructor creates and initializes a new RIV::SnmpAccess object.

Constructor

```
new($rivSession)
```

Parameters

$rivSession
Specifies a blessed reference to either a RIV::App or RIV::Agent object. More specifically, this is a RIV::App or RIV::Agent application object returned in a previous call to the RIV::App or RIV::Agent constructor.

Description

The RIV::SnmpAccess constructor creates and initializes a new RIV::SnmpAccess session object that must be a blessed reference to either a RIV::App or RIV::Agent object.

You can create only one RIV::SnmpAccess session object in any Perl application. If multiple domains are being supported (that is, multiple RIV::App objects) one of the application sessions must be used as the base for the RIV::SnmpAccess session.

Example Usage

```
$app = new RIV::App();
$snmp = new RIV::SnmpAccess('TEST', 'ncp_test');
```

Returns

Upon completion, the RIV::SnmpAccess constructor returns a RIV::SnmpAccess session object.

See Also

- “RIV::Agent Constructor” on page 188
- “RIV::App Constructor” on page 212

ASN1ToOid

The ASN1ToOid method converts the specified ASN.1 value to its corresponding OID.

Method Synopsis

```
ASN1ToOid($asn1)
```

Parameters

$asn1
Specifies the ASN.1 (Abstract Syntax Notation One) value to be converted to its corresponding object identifier (OID).

Description

The ASN1ToOid method converts the specified ASN.1 value (*$asn1*) to its corresponding OID.

Example Usage

The following example returns *$oid* as 'ifIndex':

```
$oid = $snmp->ASN1ToOid ("1.3.6.1.2.1.2.2.1.1")
```

Returns

Upon completion, the ASN1ToOid method returns an OID that corresponds to the specified ASN.1 value (*$asn1*) .

See Also

- "RIV::SnmpAccess Constructor" on page 241

AsyncSnmpGet

The AsyncSnmpGet method performs an asynchronous SNMP get operation on the specified MIB variable.

Method Synopsis

```
AsyncSnmpGet($tag, $host, $addOn,
$oid [,$instance [,$splitOutput]])
```

Parameters

$tag
Specifies a string that the AsyncSnmpGet method appends to SNMP_*$tag*. This tag is associated with the results of an SNMP get operation. For example, if you specify the string GET to the *$tag* parameter, the AsyncSnmpGet method associates the tag SNMP_GET with the results for this SNMP get operation.

$host
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid
Specifies the MIB variable for which you want to perform an asynchronous SNMP get operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, 5.3.15).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — OID, INDEX, and NAMEMIB. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The AsyncSnmpGet method performs an asynchronous SNMP get operation on the specified MIB variable (the*$oid* parameter). The AsyncSnmpGet method returns the three extra keys — OID, INDEX, and NAMEMIB — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$snmp->AsyncSnmpGet('GET', "1.2.3.4", "", "ifDescr", "2", 1);
($tag, $data) = $snmp->RIV::GetInput(-1);
```

Returns

Upon successful completion, the AsyncSnmpGet method returns the value */%varop* and the tag SNMP_ *$tag*. If the request failed, AsyncSnmpGet returns undef. The return value, along with the tag SNMP_ *$tag*, are returned in a call to RIV::GetInput.

See Also

- “RIV::GetInput” on page 177
- “RIV::SnmpAccess Constructor” on page 241
- “MaxAsyncConcurrent” on page 246

AsyncSnmpGetBulk

The AsyncSnmpGetBulk method performs an asynchronous SNMP get-bulk operation on all MIB objects in the specified MIB table.

Method Synopsis

```
AsyncSnmpGetBulk($tag, $host, $addOn,
$oidBindList, $nonRepeaters, $maxRepetitions
[,$instance [,$splitOutput]])
```

Parameters

$tag
Specifies a string that the AsyncSnmpGetBulk method appends to SNMP_*$tag*. This tag is associated with the results of an SNMP get-bulk operation. For example, if you specify the string GETBULK to the *$tag* parameter, the AsyncSnmpGetBulk method associates the tag SNMP_GETBULK with the results for this SNMP get-bulk operation.

$host
Specifies a valid a host IP address.

$addOn
Specifies the suffix to the community string.

$oidBindList
Specifies a reference to an array that contains the MIB variables for which you want to perform an asynchronous SNMP get-bulk operation. The following is an example of an array that contains two MIB variables:

```
$oidBindList = \@oids;
where,
@oids = ('sysDescr', 'ifIndex');
```

$nonRepeaters
Specifies the number of MIB variables at the start of the list of @oids that return a single value. In the previous example, the @oids list contains two MIB variables: sysDescr and ifIndex. Only the

sysDescr MIB variable returns a single value. Thus, this parameter would be set to the value 1 for the previous example.

$maxRepetitions
Specifies the number of MIB variable values in the table to be returned. For example, if you specify the value 2 to the *$maxRepetitions* parameter, the AsyncSnmpGetBulk method returns only the values for the first two MIB variables in the table. To return values for all MIB variables in the table, specify a large number for this parameter.

This parameter is relevant for MIB variables that return a table, for example, ifIndex.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, 5.3.15).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — OID, INDEX, and NAMEMIB. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The AsyncSnmpGetBulk method performs an asynchronous SNMP get-bulk operation on all MIB objects specified in *$oidBindList*. The SnmpGetBulk method returns the three extra keys — OID, INDEX, and NAMEMIB — only if the *$splitOutput* parameter is set to true (1).

Notes

The parameters *$nonRepeaters* and *$maxRepetitions* must be defined. No default values are specified for these parameters.

Example Usage

```
my @oids=
('sysDescr', 'sysContact', 'sysUpTime', 'ipInReceives',
'ipOutRequests', 'ipOutDiscards', 'ipForwDatagrams', 'tcpCurrEstab',
'ifDescr');

$snmp->AsyncSnmpGetBulk("GETBULK", "1.2.3.4", "", \@oids, 8, 100);

($tag, $data) = RIV::GetInput(-1);
```

Returns

Upon completion, the AsyncSnmpGetBulk method returns a reference to an array of *varops* and the tag SNMP_*$tag*. If the request failed, AsyncSnmpGetBulk returns undef. The return value, along with the tag SNMP_*$tag*, are returned in a call to RIV::GetInput.

See Also

- “RIV::GetInput” on page 177
- “RIV::SnmpAccess Constructor” on page 241
- “MaxAsyncConcurrent” on page 246

AsyncSnmpGetNext

The AsyncSnmpGetNext method performs an asynchronous SNMP get-next operation on the specified MIB variable.

Method Synopsis

```
AsyncSnmpGetNext($tag, $host, $addOn,
$oid [,$instance [,$splitOutput]])
```

Parameters

$tag
Specifies a string that the AsyncSnmpGetNext method appends to SNMP_*$tag*. This tag is associated with the results of an SNMP get-next operation. For example, if you specify the string GETNEXT to the *$tag* parameter, the AsyncSnmpGetNext method associates the tag SNMP_GETNEXT with the results for this SNMP get-next operation.

$host
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid
Specifies the MIB variable for which you want to perform an asynchronous SNMP get-next operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, 5.3.15).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — OID, INDEX, and NAMEMIB. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The AsyncSnmpGetNext method performs an asynchronous SNMP get-next operation on the specified MIB variable (*$oid*). The AsyncSnmpGetNext method returns the three extra keys — OID, INDEX, and NAMEMIB — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$snmp->AsyncSnmpGetNext('GETNEXT', "1.2.3.4", "", "ifDescr");

($tag, $data) = RIV::GetInput(-1);
```

Returns

Upon successful completion, the AsyncSnmpGetNext method returns a reference to an array of *varops* and the tag SNMP_*$tag*. If the request failed, AsyncSnmpGetNext returns undef. The return value, along with the tag SNMP_*$tag*, are returned in a call to RIV::GetInput.

See Also

- "RIV::GetInput" on page 177

- “RIV::SnmpAccess Constructor” on page 241
- “MaxAsyncConcurrent” on page 246

GetMibHash

The GetMibHash method gets the entire MIB tree.

Method Synopsis

```
GetMibHash()
```

Parameters

None

Description

The GetMibHash method gets the entire MIB tree by browsing the files that exist in the $NCHOME/mibs directory.

Example Usage

```
my %tree=$snmp->GetMibHash();
```

The hash list has keys associated with the MIB variables and values that correspond to the ASN.1 values.

Returns

Upon completion, the GetMibHash method returns the complete MIB tree constructed as a result of browsing the files that reside in the $NCHOME/mibs directory.

See Also

- “RIV::SnmpAccess Constructor” on page 241

MaxAsyncConcurrent

The MaxAsyncConcurrent package variable sets the maximum number of concurrent asynchronous requests.

Variable Synopsis

```
$RIV::SnmpAccess::MaxAsyncConcurrent
```

Description

The MaxAsyncConcurrent package variable sets the maximum number of concurrent asynchronous requests. The default is ten concurrent asynchronous requests. The value of this variable is used when the first asynchronous request is executed. Thereafter, any changes to this package variable are ignored.

You use this package variable with the following asynchronous methods:

- AsyncSnmpGet
- AsyncSnmpGetNext
- AsyncSnmpGetBulk

See Also

- “AsyncSnmpGet” on page 242
- “AsyncSnmpGetNext” on page 245
- “AsyncSnmpGetBulk” on page 243

OidToASN1

The `OidToASN1` method converts the specified OID to its corresponding ASN.1 value.

Method Synopsis

```
OidToASN1($oid)
```

Parameters

$oid
Specifies the object identifier (OID) to be converted to its corresponding ASN.1 (Abstract Syntax Notation One) value.

Description

The `OidToASN1` method converts the specified OID (*$oid*) to its corresponding ASN.1 value.

Example Usage

```
$asn1 = $snmp->OidToASN1('ifDescr');
```

Returns

Upon completion, the `OidToASN1` method returns an ASN.1 value that corresponds to the specified OID (*$oid*).

See Also

- “RIV::SnmpAccess Constructor” on page 241

SnmpGet

The `SnmpGet` method performs an SNMP `get` operation on the specified MIB variable.

Method Synopsis

```
SnmpGet($host, $addOn, $oid
[,$instance [,$splitOutput]])
```

Parameters

$host
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oid
Specifies the MIB variable for which you want to perform an SNMP `get` operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `SnmpGet` method performs an SNMP `get` operation on the specified MIB variable (*$oid*). The `SnmpGet` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
$vap = $snmp->SnmpGet("1.2.3.4", "", "ifDescr", 1);
print "$vap->{ASN1}, $vap->{VALUE}", "\n";
```

Returns

Upon completion, the `SnmpGet` method returns */%varop*, where the *%varop* keys are `ASN1` and `VALUE`.

See Also

- "RIV::SnmpAccess Constructor" on page 241

SnmpGetBulk

The `SnmpGetBulk` method performs an SNMP `get-bulk` operation on all MIB objects in the specified MIB table.

Method Synopsis

```
SnmpGetBulk($host, $addOn, $oidBindList,
$nonRepeaters, $maxRepetitions
[,$instance [,$splitOutput]])
```

Parameters

$host
Specifies a valid host IP address.

$addOn
Specifies the suffix to the community string.

$oidBindList
Specifies a reference to an array that contains the MIB variables for which you want to perform an SNMP `get-bulk` operation. The following is an example of an array that contains two MIB variables:

```
$oidBindList = \@oids;
where,
@oids = ('sysDescr', 'ifIndex');
```

$nonRepeaters
Specifies the number of MIB variables at the start of the list of `@oids` that return a single value. In the previous example, the `@oids` list contains two MIB variables: `sysDescr` and `ifIndex`. Only the

sysDescr MIB variable returns a single value. Thus, this parameter would be set to the value 1 for the previous example.

$maxRepetitions
Specifies the number of MIB variable values in the table to be returned. For example, if you specify the value 2 to the *$maxRepetitions* parameter, the SnmpGetBulk method returns only the values for the first two MIB variables in the table. To return values for all MIB variables in the table, specify a large number for this parameter.

This parameter is relevant for MIB variables that return a table, for example, ifIndex.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, 5.3.15).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — OID, INDEX, and NAMEMIB. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The SnmpGetBulk method performs an SNMP get-bulk operation on all MIB objects specified in *$oidBindList*. The SnmpGetBulk method returns the three extra keys — OID, INDEX, and NAMEMIB — only if the *$splitOutput* parameter is set to true (1).

Notes

The parameters *$nonRepeaters* and *$maxRepetitions* must be defined. No default values are specified for these parameters.

Example Usage

```
my @oids=('sysDescr','sysContact','sysUpTime','ipInReceives',

'ipOutRequests','ipOutDiscards','ipForwDatagrams','tcpCurrEstab',
'ifDescr');

($vap) = $snmp->SnmpGetBulk("1.2.3.4", "", \@oids, 8, 100);
```

Returns

Upon completion, the SnmpGetBulk method returns a reference to a result array. Each element of the result array is a %varop hash.

See Also

- “RIV::SnmpAccess Constructor” on page 241

SnmpGetNext

The SnmpGetNext method performs an SNMP get-next operation on the specified MIB variable.

Method Synopsis

```
SnmpGetNext($host, $addOn, $oid
[, $instance [,$splitOutput]])
```

Parameters

$host
Specifies a valid IP address.

$addOn
Specifies the suffix to the community string.

$oid
Specifies the MIB variable for which you want to perform an SNMP `get-next` operation.

$instance
Specifies the start of the MIB subtree to retrieve. You must specify *$instance* as an ASN1 string (for example, `5.3.15`).

This parameter is optional.

$splitOutput
Specifies a value of true or false. If set to true (1) returns three extra keys — `OID`, `INDEX`, and `NAMEMIB`. The default is false (0), that is, does not return the three extra keys.

This parameter is optional.

Description

The `SnmpGetNext` method performs iterative SNMP `get-next` operations on the specified host (*$nodeIP*) for the MIB table starting at the specified MIB variable (*$oid*). The `SnmpGetNext` method returns the three extra keys — `OID`, `INDEX`, and `NAMEMIB` — only if the *$splitOutput* parameter is set to true (1).

Example Usage

```
my ($vap) = $snmp->SnmpGetNext("1.2.3.4", "", "ifDescr");
```

Returns

Upon completion, the `SnmpGetNext` method returns a reference to a result array. Each element of the result array is a *%varop* hash.

See Also

- “RIV::SnmpAccess Constructor” on page 241

SnmpWalk

The `SnmpWalk` method performs an SNMP `walk` operation on the specified device, starting at the specified MIB variable.

Method Synopsis

```
SnmpWalk($host, $addOn, $oid,
$ignoreInstanceFilters)
```

Parameters

$host
Specifies a valid IP address.

$addOn
Specifies the suffix to the community string (usually an empty string).

$oid
Specifies the MIB variable from which to start the SNMP `walk` operation.

ignoreInstanceFilters
If instance filters have been configured for the SNMP helper, determines whether the SNMP `walk` operation returns only the filtered instances or all data. Pass one of the following values to this parameter:

- 0 -- Applies filters to the SNMP `walk` operation.
- 1 -- Ignores any filters.

Description

The `SnmpWalk` method performs an SNMP `walk` operation on a given device, starting at the specified MIB variable (*$oid*). The `SnmpWalk` method returns either only filtered instances or all data, depending on the value passed to the *$ignoreInstanceFilters* parameter.

Example Usage

```
my ($results, $status) = $snmp->SnmpWalk("1.2.3.4", "", "ifTable", 1);
```

Returns

Upon successful completion, the `SnmpWalk` method returns a 2-element array. The first element is a reference to an array of returned SNMP data. Each element has the same format as returned by the `SnmpGetNext` method. The second element is a hash of additional data. This hash can include the following fields:

- *m_InstanceFiltered* -- If present, this field indicates that instance filtering was applied to the returned results.
- *m_ErrorStatus* -- If present, this field indicates any error status detected by the SNMP helper while trying to process the query.

See Also

- “SnmpGetNext” on page 249
- “RIV::SnmpAccess Constructor” on page 241

SplitOidAndIndex

The `SplitOidAndIndex` method converts the full ASN.1 value into its index and the base OID.

Method Synopsis

```
SplitOidAndIndex($fullASN1)
```

Parameters

$fullASN1
Specifies the complete ASN.1 (Abstract Syntax Notation One) value to be split.

Description

The `SplitOidAndIndex` method splits the specified ASN.1 value (*$fullASN1*) into its index and the base OID (object identifier).

Example Usage

The following call to SplitOidAndIndex passes an ASN.1 value of 1.3.6.1.2.1.2.2.1.1 to the *$fullASN1* parameter:

```
($baseOid, $indexOid, $baseOidName) = $snmp->SplitOidAndIndex($fullASN1);
```

The previous call returns the following values:

- *$baseOID*=1.3.6.1.2.1.2.2.1.1
- *$indexOID*=0
- *$baseOidName*=ifDescr

Returns

Upon completion, the SplitOidAndIndex method returns an array with three elements:

- The base OID.
- The index.
- The name of the base OID.

See Also

- “RIV::SnmpAccess Constructor” on page 241

Chapter 11. NCP Modules Reference

Each NCP module provides constructors and methods used in the Perl scripts that you implement to perform operations on NCIM topology databases and NCIM domains.

To implement Perl scripts using the NCP modules, you must be familiar with the constructors and methods that each module provides. These constructors and methods are described in manual (reference) page format.

The following list identifies the NCP modules:

- `NCP::DBI_FACTORY`
- `NCP::Domain`

NCP::DBI_Factory module reference

The `NCP::DBI_Factory` module provides an interface to make it easier to use the standard Perl DBI module to perform operations on NCIM topology databases.

The `NCP::DBI_Factory` module provides a method used to create a standard DBI handle used in subsequent calls to some of the methods that perform operations on NCIM topology databases.

Use of the methods that the `NCP::DBI_Factory` module provides assumes that you understand how to use the standard Perl DBI module and that you are familiar with NCIM topology databases.

See *IBM Tivoli Network Manager Reference* for information on NCIM topology databases.

Each of the `NCP::DBI_Factory` module methods is described in manual (reference) page format.

NCP::DBI_Factory module synopsis

The `NCP::DBI_Factory` module synopsis shows how to make calls to some of the NCIM database operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the NCIM database operation methods. The reference (man) pages provide the details.

```
# Load the NCP::DBI_Factory module.
use NCP::DBI_Factory;

# Get the database login details from DbLogins.NCOMS.cfg, or,
# failing that, from DbLogins.cfg.
     my %typicalParameters = (
                          domain => "NCOMS",
                          dbid => "NCIM",
                          );

# Call the createDbHandle method to obtain the DBI handle. In this call,
# pass the %typicalParameters hash parameter.
my $dbh = NCP::DBI_Factory::createDbHandle(%typicalParameters);

my %explicitParams = (
                dbname => "ncim",
                server => "db2",
                schema => "ncim",
                host => "192.168.1.1",
                username => "dbuser",
                password => "dbpassword",
                port => 50000 # optional
              );

# Call the createDbHandle method to obtain a second DBI handle. In this call,
# pass the %explicitParameters hash parameter.
my $otherDbh = NCP::DBI_Factory::createDbHandle(%explicitParams);

# Declare variables that the insert_row and insert_auto_inc_row methods use.
my $tableName = "entityNameCache";
my $name = "entity1";
```

```
# Declare a hash that the insert_row and insert_auto_inc_row methods use.
# Note: The string in $name will automatically be quoted.
my %row = (
    entityName => $name,
    domainMgrId => 1
);

# Call the insert_row method to insert a row into a database table
# called entityNameCache.
NCP::DBI_Factory::insert_row($dbh, $tableName, \%row)
    or print "Insert failed ", $dbh->errstr "\n";

    my $autoIncColumnName = "entityId";

# Call the insert_auto_inc_row method to insert a row into a database table
# called entityNameCache. This table has an auto incremented column called
# entityId.
my $newId = NCP::DBI_Factory::insert_auto_inc_row(
    $dbh, $tableName, \%row, $autoIncColumnName)
    or print "Insert failed ", $dbh->errstr "\n";

# Set up the variabloes to use in the calls to the
# prepare_insert_auto_inc and execute_insert_auto_inc methods.
my @columnName = ["entityName","domainMgrId"];
my @values = ["entity2",2];
my $sth = NCP::DBI_Factory::prepare_insert_auto_inc(
    $dbh, $tableName, $autoIncColumnName, @columnNames)
    or print "Prepared failed ", $dbh->errstr, "\n";

my $newId2 = NCP::DBI_Factory::execute_insert_auto_inc(
    $dbh, $sth, @values)
    or print "Insert failed ", $dbh->errstr "\n";

$sth->finish();

# Commit the changes to the NCIM topology database by calling
# the Perl DBI module commit method. Otherwise, call the Perl DBI
# rollback method to undo the most recent series of uncommitted
# database changes.
if ($happy)
{
    $dbh->commit();
}
else
{
    $dbh->rollback();
}

# Identify the current schema by calling the schema method.
# Call the tables method to return a sorted array of table
# and view names for the current schema.
# Call the describeTable method to return a sorted array
# of upper case field names for the specified table
# in the current schema.
my $schema = NCP::DBI_Factory::schema(%typicalParameters);
my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                         schema => $schema,
                                         %typicalParameters);
foreach my $table (@tableList)
{
my @fields = NCP::DBI_Factory::describeTable(
                                         $table,
                                         dbh => $dbh,
                                         schema => $schema);

# Disconnect and clean.
NCP::DBI_Factory::finish( $dbh );
NCP::DBI_Factory::finish( $otherDbh );
```

createDbHandle

The `createDbHandle` method creates a standard DBI handle, connected to the requested NCIM topology database. This DBI handle is used in subsequent calls to some of the other `NCP::DBI_Factory` module methods.

Method Synopsis

```
NCP::DBI_Factory::createDbHandle(%typicalParameters)
NCP::DBI_Factory::createDbHandle(%explicitParameters)
```

Parameters

%typicalParameters

Specifies a hash that contains the key/value pairs necessary for `createDbHandle` to access information from one of the following files in order to create a DBI handle:

- `DbLogins.cfg` — Specifies the standard database log-ins configuration file.
- `DbLogins.`*domain*`.cfg` — Specifies a domain-specific database log-ins configuration file where *domain* identifies a domain (for example, `DbLogins.NCOMS.cfg`).
- Custom file — Specifies an optional custom database log-ins configuration file. This file is expected to have the same format as `DbLogins.cfg` and `DbLogins.`*domain*`.cfg`.

The following table identifies the key/value pairs in this hash:

Hash key	Description
`domain`	Specifies the name of the domain used to identify whether a `DbLogins.`*domain*`.cfg` file exists in the `$NCHOME/etc/precision` directory. The following example shows a possible value for this key: `domain => "NCOMS"` In this example, the `createDbHandle` method would look for a file called `DbLogins.NCOMS.cfg` in the `$NCHOME/etc/precision` directory. If a `DbLogins.`*domain*`.cfg` file does not exist, `createDbHandle` looks for the `DbLogins.cfg` file. This is a required key/value pair.
`dbid`	Specifies the logical name for the NCIM topology database to which you want to connect. Each NCIM topology database has a unique logical name specified in the `DbLogins.cfg`, `DbLogins.`*domain*`.cfg`, or custom database log-ins configuration file. The following example shows a possible value for this key: `dbid => "ncim"` In this example, the value `ncim` specifies the logical name for this connection to the NCIM topology database. The `createDbHandle` method uses `dbid` to locate the appropriate section of the database log-ins configuration file. This is a required key/value pair. **Note:** The `dbid` key/value pair maps to the `m_DbId` field in the database log-ins configuration file.

Hash key	Description
dbfile	Specifies the name of the custom database log-ins configuration file. If you specify the optional dbfile key/value pair, the createDbHandle method would look for the specified custom file in the $NCHOME/etc/precision directory.

%explicitParameters

Specifies a hash that contains the key/value pairs necessary to create a DBI handle. In this case, createDbHandle does not obtain the necessary values from a file as is the case for the *%typicalParameters* hash parameter. Instead, all of the necessary values are explicitly specified. (Typically, an application would obtain these values from the command line.) The following table identifies the key/value pairs in this hash. All key/value pairs listed in the table are required, except for port, which is optional.

Hash key	Description
dbname	Specifies the name of the NCIM topology database to which you want to connect. The following example shows a possible value for this key: `dbname => "ncim"` In this example, the value ncim specifies that you want to connect to the NCIM topology database.
server	Specifies a string that identifies the type of database associated with the database name specified in the dbname key. The following list identifies the possible values for this key: • oracle — Specifies the Oracle database. • db2 — Specifies the Db2 database. The following example shows a database type of Db2: `server => "db2"`
schema	Specifies the name of the schema to access in the database specified in the dbname key. The following example shows a possible value for this key: `schema => "ncim"`
host	Specifies the address of the host computer on which the specified NCIM topology database resides. The following example shows a possible value for this key: `host => "192.168.1.1"`
username	Specifies the name of the user who has access to the specified NCIM topology database. The following example shows a possible value for this key: `username => "dbuser"`
password	Specifies the password of the user who has access to the specified NCIM topology database. The following example shows a possible value for this key:

Hash key	Description
	`password => "dbpassword"`
`port`	Specifies an optional key that identifies the port associated with the address specified in `host`. The following example shows a possible value for this key: `port => "3406"`

Description

The `createDbHandle` method creates a standard DBI (Database Interface) handle to be used in subsequent calls to some of the other `NCP::DBI_Factory` methods. This DBI handle contains the information needed to connect to the requested NCIM topology database.

The `createDbHandle` method accepts the following hash parameters:

- *%typicalParameters* — This hash provides the `domain` and `dbid` key/value pairs. Optionally, this hash can provide a `dbfile` key/value pair. Given this information, the `createDbHandle` method:
  - Reads and parses one of these files that resides in the `$NCHOME/etc/precision` directory: `DbLogins.cfg` (the default), `DbLogins.`*`domain`*`.cfg`, or an optional custom database login-ins configuration file.
  - Uses the `dbid` key/value pair to locate the database entry of interest in the specified database log-ins configuration file.
  - Connects to the specified NCIM topology database.
  - Sets the context to the schema associated with the specified NCIM topology database.
- *%explicitParameters* — This hash provides all of the required information from the command line. Given this information, the `createDbHandle` method:
  - Connects to the specified NCIM topology database.
  - Sets the context to the schema associated with the specified NCIM topology database.

When reading from a database log-ins configuration file, `createDbHandle` can override any values from the file if you explicitly pass them in from the command line. The following table provides the available override options and their mappings to the fields in the database log-ins configuration file:

Override option	Description
`dbfile`	Specifies an optional override for the `DbLogins.cfg`database log-ins configuration file. This file is expected to have the same format as `DbLogins.cfg` and `DbLogins.`*`domain`*`.cfg`.
`dbname`	Specifies the name of the database. If specified on the command line, this option overrides the value specified for the `m_DbName` field in the database log-ins configuration file.
`server`	Specifies a string that identifies the type of database associated with the database name specified in the `dbname` option. The following list identifies the possible values for the `server` option: • `oracle` — Specifies the Oracle database. • db2 — Specifies the Db2 database. This option overrides the value specified for the `m_Server` field in the database log-ins configuration file.

Override option	Description
schema	Specifies the name of the schema to access in the specified database. This option overrides the value specified for the m_Schema field in the database log-ins configuration file.
host	Specifies the address of the host computer on which the specified NCIM topology database resides. This option overrides the value specified for the m_Hostname field in the database log-ins configuration file.
username	Specifies the name of the user who has access to the specified NCIM topology database. This option overrides the value specified for the m_Username field in the database log-ins configuration file.
password	Specifies the password of the user who has access to the specified NCIM topology database. This option overrides the value specified for the m_Password field in the database log-ins configuration file.
port	Specifies the port associated with the address specified in host. This option overrides the value specified for the m_PortNum field in the database log-ins configuration file.

Notes

To ensure that the createDbHandle method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code example illustrates a typical call to the createDbHandle method using the *%typicalParameters* hash parameter:

```
# Set up the hash list to contain the domain and
# database ID.

my %typicalParameters = (
                            domain => "NCOMS",
                            dbid => "NCIM"
                            );

# Call the createDbHandle method passing to it the previously
# set up hash list. In this case, createDbHandle knows that the
# information it needs to create the DBI handle resides in a file.
# The createDbHandle method returns the DBI to the $dbh variable.
#

my $dbh = NCP::DBI_Factory::createDbHandle(%typicalParameters);
```

The following code example illustrates a typical call to the createDbHandle method using the *explicitParams* parameter:

```
# Set up the hash list to contain the information necessary to create
# the DBI handle without reading a database log-ins configuration file.

my %explicitParams = (
                   dbname => "ncim",
                   server => "db2",
                   schema => "ncim",
                   host => "9.180.209.24",
                   username => "batman",
                   password => "robin",
                   port => 3406 # This is an optional element.
                 );
```

```
# Call the createDbHandle method passing to it the previously set up
# hash list that contains the information necessary to create the
# DBI handle. The createDbHandle method returns the DBI handle to
# the $otherDbh variable.
my $otherDbh = NCP::DBI_Factory::createDbHandle(%explicitParams);
```

Returns

Upon completion, the `createDbHandle` method returns a standard DBI handle associated with the requested NCIM topology database.

See Also

- "schema" on page 269

describeTable

The `describeTable` method returns a sorted array of uppercase field names for the specified table or view.

Method Synopsis

```
NCP::DBI_Factory::describeTable($tableName, %dbhschema)

NCP::DBI_Factory::describeTable($tableName, %typicalParameters)
```

Parameters

$tableName
Specifies the name of the database table that is of interest. Because the different databases that the `DBI_Factory` module supports use different cases for table names, supply the table name in mixed case. For example, if the table name is `entitynamecache`, then the mixed case equivalent is `entityNameCache`.

In either case, the `describeTable` method internally converts the specified database table name to upper or lower case as required.

%dbhschema
Specifies a hash that contains the following keys:

- dbh — Specifies an existing DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.
- `schema` — Specifies the schema that contains the database table name specified in the *$tableName* parameter. Typically, this schema name is obtained in a call to the `schema` method.

%typicalParameters
Specifies the same hash parameter accepted by the `createDbHandle` method.

Description

The `describeTable` method returns a sorted array of uppercase field names for the database table or view specified in the *$tableName* parameter. For full portability, pass this table name in mixed case to the *tableName* parameter. The `describeTable` method:

- Converts the table name to upper case for Oracle and Db2 databases, since these databases require upper case table names. For example, the table name `entityNameCache` would be converted to `ENTITYNAMECACHE`.

If you specify the *%typicalParameters* hash instead of the *%dbhschema* hash, `describeTable` calls `createDbHandle` to create a new DBI handle. The schema associated with this newly created handle is

identified by the dbid key/value pair and this schema is expected to contain the table specified in the *$tableName* parameter.

Notes

The NCP::DBI_Factory module supports Db2 and Oracle databases. In both these databases, tables and field names are case insensitive with regard to SQL statements. However, Db2 and Oracle return field names in table rows in uppercase.

Example Usage

The following code example illustrates a typical call to the describeTable method using the *%dbhschema* hash parameter. The code example also shows a call to the tables method:

```
# List the tables in schema $schema without creating a new DBI handle.
my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                         schema => $schema);
 foreach my $table (@tableList)
  {
     my @fields = NCP::DBI_Factory::describeTable(
                                          $table,
                                          dbh => $dbh,
                                          schema => $schema);
    }
```

Returns

Upon completion, the describeTable method returns a sorted array of uppercase field names for the specified database table or view.

See Also

- “createDbHandle” on page 255
- “schema” on page 269
- “tables” on page 272

execute_insert_auto_inc

The execute_insert_auto_inc method executes an auto-incremented column statement handle prepared by the prepare_insert_auto_inc method.

Method Synopsis

```
NCP::DBI_Factory::execute_insert_auto_inc($dbHandle,$statementHandle,
 $values)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the createDbHandle method. This handle supplies the context for connecting to the specified NCIM topology database.

$statementHandel
Specifies the statement handle returned in a previous call to the prepare_insert_auto_inc method.

$values
Specifies the values to be executed.

Description

The execute_insert_auto_inc method executes an auto-incremented column statement handle prepared by the prepare_insert_auto_inc method.

Notes

To ensure that the execute_insert_auto_inc method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code example illustrates a typical call to the execute_insert_auto_inc method:

```
my $newId2 = NCP::DBI_Factory::execute_insert_auto_inc($dbh, $sth, @values)
    or print "Insert failed ", $dbh->errstr "\n";
```

Returns

Upon completion, the execute_insert_auto_inc method returns the new auto-incremented value.

See Also

- "createDbHandle" on page 255
- "prepare_insert_auto_inc" on page 268

extractCmdLineOptions

The extractCmdLineOptions method allows database login options specified on the command line to be provided in a common format.

Method Synopsis

```
NCP::DBI_Factory::extractCmdLineOptions([$prefix])
```

Parameters

$prefix

An optional parameter that specifies a prefix used to allow other similar database login options to be supplied for multiple database connections. Examples of such prefixes include ncim_, ncmonitor_, and ncpoller_.

Description

The extractCmdLineOptions method allows database login options specified on the command line to be provided in a common format. This method accepts the same database login options as the createDbHandle method:

- dbfile
- server
- dbname
- schema
- host
- username
- password

- port

The extractCmdLineOptions method can also take an optional *$prefix* parameter that specifies similar database login options other than the previously listed options. This optional parameter allows Perl scripts to handle multiple sets of database login options by calling the extractCmdLineOptions method multiple times.

Notes

The extractCmdLineOptions method removes the database login options that it processes and returns in the hash from the @ARGV array. However, any options that do not get processed and returned in the hash remain in the @ARGV array.

Use the extractCmdLineOptions method to process database login options from the command line. Use the extractHashRefOptions method to process database login options from a hash reference.

Example Usage

You can call the extractCmdLineOptions method with or without the *$prefix* parameter.

Calling extractCmdLineOption without the $prefix parameter

The following code example illustrates a call to the extractCmdLineOptions method without the use of the *$prefix* optional parameter. The example declares a variable called *$optionsHashRef* to store the reference to the hash returned by extractCmdLineOptions:

```
my $optionsHashRef = NCP::DBI_Factory::extractCmdLineOptions();
```

Assume that the previous code example is contained in a Perl script called dboptions.pl. Consider this script executed with the password , host, and whatever database login options:

```
dboptions.pl -password tom -host dick -whatever harry
```

The extractCmdLineOptions returns a reference to a hash in *$optionsHashRef* as follows:

```
$optionsHashRef = { password => "tom", host => "dick" }
```

The database login option — ('-whatever', 'harry') — remains in the @ARGV array because the extractCmdLineOptions method could not process it without the *$prefix* optional parameter.

Calling extractCmdLineOption with the $prefix parameter

The following code example illustrates multiple calls to the extractCmdLineOptions method with the use of the *$prefix* optional parameter:

```
my $generic =
NCP::DBI_Factory::extractCmdLineOptions();
my $ncimSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncim_")|| $generic;
my $ncmonitorSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncmonitor_") || $generic;
my $ncpollerSpecific =
NCP::DBI_Factory::extractCmdLineOptions("ncpoller_") || $generic;
```

Assume that the previous code example is contained in a Perl script called dboptionsuseprefix.pl. Consider this script executed with the password and ncpoller_password database login options:

```
dboptionsuseprefix.pl -password "ncim" -ncpoller_password "ncpoller"
```

The extractCmdLineOptions returns references to hashes in *$ncimSpecific*, *$ncmonitorSpecific*, and *$ncpollerSpecific* as follows:

```
$ncimSpecific = { password => "ncim" };
$ncmonitorSpecific = { password => "ncim" };
$ncpollerSpecific = { password => "ncpoller" };
```

The following list further explains how these calls to `extractCmdLineOptions` work:

- The first call to `extractCmdLineOptions` (without the optional *$prefix* parameter) processes the `-password "ncim` database login option and returns a reference to a hash that contains `{ password => "ncim" }`.
- The second call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncim_` is specified, then process it and return the appropriate value in the hash reference. Otherwise, return `{ password => "ncim" }` to the hash reference. In this case, the right side of the logical OR is true.
- The third call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncmonitor_` is specified, then process it and return the appropriate value in the hash reference. Otherwise return `{ password => "ncim" }` to the hash reference. In this case, the right side of the logical OR is true.
- The fourth call to `extractCmdLineOptions` sets up a logical OR operation. If a database login option beginning with the prefix `ncpoller_` is specified, then process it and return the appropriate value in the hash reference. Otherwise return `{ password => "ncim" }` to the hash reference. In this case, the left side of the logical OR is true and so `password => "ncpoller` is returned.

Returns

Upon completion, the `extractCmdLineOptions` method returns a reference to a hash that contains the extracted database login options and values in key/value format. If no database login options were specified, the `extractCmdLineOptions` method returns `undef`.

See Also

- “createDbHandle” on page 255
- “extractHashRefOptions” on page 263

extractHashRefOptions

The `extractHashRefOptions` method extracts the database login options from the specified hash reference.

Method Synopsis

```
NCP::DBI_Factory::extractHashRefOptions($originalHashRef [,$prefix])
```

Parameters

$originalHashRef
Specifies a reference to the original hash that contains the database login options.

$prefix
An optional parameter that specifies a prefix used to allow other similar database login options to be supplied for multiple database connections. Examples of such prefixes include `ncim_`, `ncmonitor_`, and `ncpoller_`.

Description

The `extractHashRefOptions` method extracts the database login options from the hash reference specified in the *$originalHashRef* parameter. This method accepts the same database login options as the `createDbHandle` method:

- `dbfile`
- `server`
- `dbname`

- schema
- host
- username
- password
- port

The extractHashRefOptions method can also take an optional *$prefix* parameter that specifies similar database login options other than the previously listed options. This optional parameter allows Perl scripts to handle multiple sets of database login options by calling the extractHashRefOptions method multiple times.

Notes

The extractHashRefOptions method does not remove the key/value pairs from the hash reference specified in the *$originalHashRef* parameter.

Use the extractHashRefOptions method to process database login options from a hash reference. Use the extractCmdLineOptions method to process database login options from the command line or DbLogins.cfg file.

Example Usage

The following code example sets up a hash reference and then makes two calls to the extractHashRefOptions method:

```
my %original =
{ password => "topsecret", ncpoller_password => "classified, foo => "bar" };

my $generic = NCP::DBI_Factory::extractHashRefOptions(\%original);
my $ncpoller = NCP::DBI_Factory::extractHashRefOptions(\%original, "ncpoller_");
```

The following further explains how these calls to extractHashRefOptions work:

- The first call to extractHashRefOptions extracts the -password => "topsecret" database login option and returns it to *$generic*. This call to extractHashRefOptions cannot extract the other two options because this call did not specify ncpoller_ or foo_ in the optional *$prefix* parameter. The -password => "topsecret" option remains in the %original hash reference.
- The second call to extractHashRefOptions extracts the -password => "classified" database login option and returns it to *$ncpoller*. This call to extractHashRefOptions extracts -password => "classified" because of the ncpoller_ prefix passed to the *$prefix* parameter. The -password => "classified" option remains in the %original hash reference.
- The foo => "bar" option is not extracted and remains in the %original hash reference.

Returns

Upon completion, the extractHashRefOptions method returns a reference to a hash that contains the extracted database login options and values in key/value format. If no database login options were specified, the extractHashRefOptions method returns undef.

See Also

- “createDbHandle” on page 255
- “extractCmdLineOptions” on page 261

finish

The `finish` method disconnects and cleans the specified database handle.

Method Synopsis

```
NCP::DBI_Factory::finish( $dbHandle );
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the createDbHandle method. This handle supplies the context for finishing the connection for the specified NCIM topology database.

Description

The `finish` method disconnects and cleans the specified database handle.

Example usage

```
NCP::DBI_Factory::finish( $dbHandle );
```

Returns

Upon completion, the `finish` method returns no value.

See Also

- "createDbHandle" on page 255

insert_auto_inc_row

The `insert_auto_inc_row` method inserts a row into the specified table that has an auto-increment column.

Method Synopsis

```
NCP::DBI_Factory::insert_auto_inc_row($dbHandle,$tableName,
$tableRow, $autoIncColumnName)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
Specifies the name of the table into which the `insert_auto_inc_row` method inserts the row specified in the *$tableRow* parameter.

$tableRow
Specifies a hash of scalars keyed on the column name.

$autoIncColumnName
Specifies the name of the auto-increment column in the specified table.

Description

The `insert_auto_inc_row` method inserts the row specified in the *$tableRow* parameter into the table specified in the *$tableName* parameter. The table is expected to contain the auto-increment column name specified in the `$autoIncColumnName` parameter.

Note: You can also call the DBI `rollback` interface to undo the most recent insert row change.

Notes

To ensure that the `insert_auto_inc_row` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to STDOUT.

Use the `insert_auto_inc_row` method to insert rows in tables that have an auto- increment column. Use the `insert_row` method to insert rows in tables that have do not have an auto- increment column.

Example Usage

The following code example illustrates a typical call to the `insert_auto_inc_row` method:

```
my $tableName = "entityNameCache";
    my $name = "fred";

    # Note: the string in $name will automatically be quoted
    my %row = (
        entityName => $name,
        domainMgrId => 1
    );

my $autoIncColumnName = "entityId";

    my $newId = NCP::DBI_Factory::insert_auto_inc_row(
        $dbh, $tableName, \%row, $autoIncColumnName)
        or print "Insert failed ", $dbh->errstr "\n";

    # Changes only take effect when this is called
    if ($happy)
    {
        $dbh->commit();
    }
    else
    {
        $dbh->rollback();
    }
```

Returns

Upon completion, the `insert_auto_inc_row` method returns the new auto-increment value, provided that the row could be uniquely identified by the fields that the `insert_auto_inc_row` method just inserted into the specified table.

See Also

- “createDbHandle” on page 255
- “insert_row” on page 266

insert_row

The `insert_row` method inserts a row into the specified table.

Method Synopsis

```
NCP::DBI_Factory::insert_row($dbHandle,$tableName, $tableRow)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
Specifies the name of the table into which the `insert_row` method inserts the row specified in the *$tableRow* parameter.

$tableRow
Specifies a hash of scalars keyed on the column name.

Description

The `insert_row` method inserts the row specified in the *$tableRow* parameter into the table specified in the *$tableName* parameter. The `insert_row` method automatically interpolates any strings in *$tableRow* into double-quoted strings.

Note: You can also call the DBI `rollback` interface to undo the most recent insert row change.

Notes

To ensure that the `insert_row` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to STDOUT.

Use the `insert_row` method to insert rows in tables that have do not have an auto- increment column. Use the `insert_auto_inc_row` method to insert rows in tables that have an auto- increment column.

Example Usage

The following code example illustrates a typical call to the `insert_row` method:

```
my $tableName = "entityNameCache";
    my $name = "fred";

    # Note: the string in $name will automatically be quoted
    my %row = (
        entityName => $name,
        domainMgrId => 1
    );

    NCP::DBI_Factory::insert_row($dbh, $tableName, \%row)
        or print "Insert failed ", $dbh->errstr "\n";

# Changes only take effect when commit is called
    if ($happy)
    {
        $dbh->commit();
    }
    else
    {
        $dbh->rollback();
    }
```

Returns

Upon completion, the `insert_row` method returns whatever the standard DBI statement handle execute method returns.

See Also

- “createDbHandle” on page 255
- “insert_auto_inc_row” on page 265

prepare_insert_auto_inc

The `prepare_insert_auto_inc` method prepares the SQL statement once so that it can be used multiple times when inserting many rows into an auto-increment column of the specified database table.

Method Synopsis

```
NCP::DBI_Factory::prepare_insert_auto_inc($dbHandle,$tableName,
$autoIncColumnName, $columnNames)
```

Parameters

$dbHandle
Specifies the DBI handle returned in a previous call to the `createDbHandle` method. This handle supplies the context for connecting to the specified NCIM topology database.

$tableName
Specifies the name of the table into which the `prepare_insert_auto_inc` method prepares the SQL statement to be inserted into multiple rows in the specified columns.

$autoIncColumnName
Specifies the name of the auto-increment column in the specified table.

$columnNames
Specifies a hash of column names.

Description

The `prepare_insert_auto_inc` method prepares the SQL statement once so that it can be used multiple times when inserting many rows into an auto-increment column of the specified database table. Use this method when inserting many rows into an auto-incremented column.

The returned SQL statement handle should be used with the `execute_insert_auto_inc` method.

Notes

To ensure that the `prepare_insert_auto_inc` method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code example illustrates a typical call to the `prepare_insert_auto_inc` method:

```
my $sth =
NCP::DBI_Factory::prepare_insert_auto_inc($dbh,
                                          $tableName,
                                          $autoIncColumnName, @columnNames)
or print "Prepared failed ", $dbh->errstr, "\n";
```

Returns

Upon completion, the `prepare_insert_auto_inc` method returns the prepared SQL statement handle.

See Also

- “createDbHandle” on page 255
- “insert_row” on page 266

schema

The schema method returns the schema name associated with the specified database.

Method Synopsis

```
NCP::DBI_Factory::schema(%typicalParameters)

NCP::DBI_Factory::schema(%explicitParameters)
```

Parameters

$typicalParameters
Specifies the same hash parameter accepted by the createDbHandle method. If you supply this hash parameter, schema obtains the value from the database log-ins configuration file.

%explicitParameters
Specifies the same hash parameter accepted by the createDbHandle method. If you supply this hash parameter, schema obtains the value from the command line.

Description

The schema method returns the name of the schema being used as follows:

- If the *%typicalParameters* hash was specified — The schema method obtains the name of the schema being used from one of these files: DbLogins.cfg, DbLogins.*DOMAIN*.cfg, or an optional custom database log-ins configuration file. If schema finds a domain-specific file, it uses that file to obtain the name of the schema. If the *schema* variable was passed to the createDbHandle method, then the schema method uses this variable to obtain the name of the schema. The *schema* variable overrides the schema information contained in any of the configuration files.
- If the *%explicitParameters* hash was specified — The schema method obtains the name of the schema from the command line. (The schema name is the value associated with the schema key in the *%explicitParameters* hash.)

Notes

To ensure that the schema method can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code fragment illustrates a typical call to the schema method using the *%typicalParams* hash parameter:

```
# Set up the hash list to contain the domain and database ID.
my %typicalParameters = (
                            domain => "NCOMS",
                            dbid => "NCIM"
                            );

# Call the schema method passing to it the previously set up hash list.
# In this case, the schema method knows that the name of the schema
# resides in a database log-ins configuration file. The schema method
# returns the name of the schema being used to the $schema variable.
#

my $schema = NCP::DBI_Factory::schema(%typicalParameters);
```

Consider the following entry in a DbLogins.cfg file. The previous call to the schema method would return a schema name of ncim (m_schema), which is associated with the database whose logical name is identified by the string NCIM (m_DbId).

```
insert into config.dbserver
(
    m_DbId,
    m_Server,
    m_DbName,
    m_Schema,
    m_Hostname,
    m_Username,
    m_Password,
    m_PortNum,
    m_EncryptedPwd,
    m_OracleService
)
values
(
    "NCIM",  -- Logical name for this connection (don't change it)
    "db2",
    "NCOMS",
    "ncim",
    "ITNMServer.company.com",
    "itnm_db_user",
    "itnm_db_password",
    50000,
    0,
    1
);
```

Returns

Upon completion, the `schema` method returns the name of the schema associated with the specified NCIM topology database.

See Also

- "createDbHandle" on page 255

setLogHandle

The `setLogHandle` method passes in the specified log handle associated with an opened file to which the `NCP::DBI_Factory` module methods can write messages.

Method Synopsis

```
NCP::DBI_Factory::setLogHandle($filehandle)
```

Parameters

$filehandle
Specifies a reference to a file handle (for example, `IO::File`) that points to an opened file to which messages can be written.

Description

The `setLogHandle` method passes in the log handle specified in the *$filehandle* parameter to an internal utility method called by the `NCP::DBI_Factory` module methods. This handle is associated with an opened file to which this internal utility method writes messages. In effect, this opened file serves as a log file that can contain debug, critical, informational, and warning type messages associated with the execution of the `NCP::DBI_Factory` module methods.

If you do not call the `setLogHandle` method, the internal utility method writes these messages to `STDOUT`.

To control the level of message reporting, call the `setLogLevel` method and specify the desired log level.

Example Usage

The following code example shows a call to the `setLogHandle` method so that messages get logged to an open file (whose associated file handle is specified in the *$logFile* local variable) rather than to STDOUT. The code example also shows a call to the `setLogLevel` method that specifies the logging of messages at the `warn` and `critical` levels.

```
.
.
.
my $logName = "$logdir/checkPing.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    NCP::DBI_Factory::setLogHandle($logFile);
.
.
.

NCP::DBI_Factory::setLogLevel("warn");
```

Returns

Upon completion, the `setLogHandle` method returns no data.

See Also

- "setLogLevel" on page 271

setLogLevel

The `setLogLevel` method sets the log level for error and message reporting.

Method Synopsis

```
NCP::DBI_Factory::setLogLevel($loglevel)
```

Parameters

$loglevel

Specifies the log level to set. The following are the valid options described in ascending order:

- `debug` — Specifies a log level in which all messages are logged.
- `info` — Specifies a log level in which informational, warning, and critical messages are logged.
- `warn` — Specifies a log level in which warning and critical messages are logged.
- `critical` — Specifies a log level in which only critical messages are logged.

Description

The `setLogLevel` method sets the log level to the option specified in the *$loglevel* parameter. The default is `debug` level. If set to a higher level, only messages with an equal or higher level will be logged. For example, at level `warn`, messages of level `info` and level `debug` will not be logged.

By default, the `NCP::DBI_Factory` module methods log messages to STDOUT. If you specify a log handle to the `setLogHandle` method, the `NCP::DBI_Factory` module methods log messages to the opened file associated with this log handle.

The `setLogLevel` method logs an appropriate message (either to STDOUT or to an opened file) if you specify an invalid log level.

Example Usage

The following code example illustrates a call to the `setLogLevel` method that specifies the logging of messages at the `warn` and `critical` levels. The code example also shows a call to the `setLogHandle` method so that these messages get logged to an open file (stored in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkPing.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    NCP::DBI_Factory::setLogHandle($logFile);
.
.
.
NCP::DBI_Factory::setLogLevel("warn");
```

Returns

Upon completion, the `setLogLevel` method returns no data.

See Also

- "setLogHandle" on page 270

tables

The `tables` method returns a sorted array of table and view names for the current schema.

Method Synopsis

```
NCP::DBI_Factory::tables(%dbhschema)

NCP::DBI_Factory::tables(%typicalParameters)
```

Parameters

%dbhschema
Specifies a hash that contains the following keys:

- dbh — Specifies an existing DBI handle returned in a previous call to the `createDbHandle` method.
- `schema` — Specifies the schema that contains the tables of interest. Typically, this schema name is obtained in a call to the `schema` method.

%typicalParameters
Specifies the same hash parameter accepted by the `createDbHandle` method.

Description

The `tables` method returns a sorted array of table and view names for the current schema.

If you specify the *%dbhschema* hash, the `tables` method:

- Uses the dbh key/value pair to identify the existing DBI handle returned in a previous call to the `createDbHandle` method. This DBI handle provides the context for connecting to the specified NCIM topology database.
- Uses the current schema specified in the `schema` key/value pair to obtain the tables of interest.
- Returns a sorted array of the table and view names for all tables associated with the current schema.

If you specify the *%typicalParameters* hash, the `tables` method:

- Creates a new DBI handle. This DBI handle provides the context for connecting to the specified NCIM topology database.
- Uses the `dbid` key/value pair to identify the current schema. For example, if the `dbid` key/value pair is `dbid => "NCIM"`, then the current schema might be called `ncim`.
- Uses the current schema identified in the `dbid` key/value pair to obtain the tables of interest.
- Returns a sorted array of the table and view names for all tables associated with the current schema.

Example Usage

The following code example illustrates a typical call to the `tables` method using the *%dbhschema* hash parameter:

```
# List the tables in schema $schema without creating a new DBI handle.
        my @tableList = NCP::DBI_Factory::tables(dbh => $dbh,
                                                 schema => $schema);
```

Returns

Upon completion, the `tables` method returns a sorted array of table and view names for the current schema.

See Also

- “createDbHandle” on page 255
- “describeTable” on page 259
- “schema” on page 269

timeStamp

The `timeStamp` method returns a timestamp in a format suitable for addition to the NCIM topology database.

Method Synopsis

```
NCP::DBI_Factory::timeStamp([$unixtimestamp])
```

Parameters

$unixtimestamp
Specifies a UNIX timestamp. This is an optional parameter. If you do not specify this parameter, the `timeStamp` method uses the current timestamp on the local host.

Description

The `timeStamp` method converts the current timestamp on the local host (or the UNIX timestamp if specified in the *unixtimestamp* parameter) to the following format that is suitable for addition to the requested NCIM topology database:

```
YYYY-MM-DD HH:MM:SS
```

where:

- *YYYY* — Specifies the year.
- *MM* — Specifies the month.
- *DD* — Specifies the day.
- *HH* — Specifies the hour.

- *MM* — Specifies the minutes.
- *SS* — Specifies the seconds.

The `timeStamp` method adds leading zeroes to any of the previous fields whose values are less than 10.

Example Usage

The following code example illustrates a call to the `timeStamp` method, specifying the current timestamp on the local host:

```
.
.
.
my $currenttime
$currenttime = timestamp();
.
.
.
```

If the current timestamp on the local host is `June 6, 2010 5:39:45` EST, the `timeStamp` method converts it to the following format that is suitable for addition to the requested NCIM topology database:

```
2010-06-04-18:39:45
```

The following code example illustrates a call to the `timeStamp` method, specifying a UNIX timestamp:

```
my $currenttime
$currenttime = timestamp(1275694785);
```

The `timeStamp` method converts this UNIX timestamp to the following format that is suitable for addition to the requested database:

```
2010-06-04-18:39:45
```

Returns

Upon completion, the `currentTimeStamp` method returns the current timestamp on the local host (or the UNIX timestamp) in the following format:

```
YYYY-MM-DD HH:MM:SS
```

toUpper

The `toUpper` method returns a copy of a hash (a single row retrieved from an NCIM database table) with all field names converted to uppercase.

Method Synopsis

```
NCP::DBI_Factory::toUpper(%rowHashRef)
```

Parameters

%rowHashRef
Specifies a hash that is a single row retrieved from an NCIM database table. The field names in this row can be specified in mixed case, lowercase, or uppercase.

Description

The `toUpper` method takes the hash (a single row retrieved from an NCIM database table) specified in the *%rowHashRef* parameter and returns a copy of this hash with all field names converted to uppercase.

The reason for providing this method is to ensure consistency across databases. Different database implementations return field names in different formats (mixed case, uppercase, or lowercase). By always converting field names to uppercase, client scripts can be made database-server-independent. To do this, pass all returned rows through this method and perform any subsequent lookup operations with uppercase field names.

The `toUpper` method drops any undefined fields in the row to promote consistent behavior across the supported databases.

Notes

The `NCP::DBI_Factory` module supports Db2 and Oracle databases. In both these databases, tables and field names are case insensitive with regard to SQL statements. However, Db2 and Oracle return field names in table rows in uppercase.

Example Usage

The following code example makes calls to methods defined in the Perl DBI module to prepare, execute, and fetch a select statement:

```
my $statement = $dbh->prepare( $selectQuery );
$statement->execute();
my $results = $statement->fetchall_arrayref({});
```

The following list provides a line-by-line explanation of the previous code example:

- The first line calls the `prepare` method to prepare the select statement specified in *$selectQuery* for later execution by the database engine. The `prepare` method returns a reference to a statement handle object in *$statement*.
- The second line uses the statement handle object returned in *$statement* to get attributes of the select statement and then invokes the `execute` method to process the prepared statement.
- The third line calls the `fetchall_arrayref` method to fetch all the data returned from the previously prepared and executed select statement. The `fetchall_arrayref` method returns to *$result*s a reference to an array that contains one reference per row.

The following code example calls the `toUpper` method to convert all field names to upper case:

```
foreach my $row (@$results)
    {
        $row = NCP::DBI_Factory::toUpper($row);
    }
```

The following list provides a line-by-line explanation of the previous code example:

- The first line sets up a `foreach` loop that iterates through the *@$result*s array.
- For each field name in the table row, call the `toUpper` method to covert the name to uppercase.

The following code example shows that fields can be safely extracted using uppercase field names:

```
foreach my $row (@$results)
    {
        my $entityId = $row->{ENTITYID};
    }
```

Returns

Upon completion, the `toUpper` method returns a copy of a hash (a single row retrieved from a database table) with all field names converted to upper case.

NCP::Domain Reference

The NCP::Domain module provides an interface to perform operations on the Network Connectivity and Inventory Model (NCIM) topology database that resides in a single Network Manager domain.

The NCP::Domain module provides a constructor that creates a new NCP::Domain object that you use to call methods that perform these tasks:

- Create an entry in the domainMgr table for this domain if one does not already exist
- Create a new domain that is a copy of an existing domain
- Remove all references to the specified domain from the domainMgr table
- Retrieve the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain
- Return the domain name for the current domain
- Pass in a log handle associated with an opened file used for logging messages
- Set the log level for error and message reporting

Use of the methods that the NCP::Domain module provides assumes that you understand concepts related to the NCIM topology database.

See *IBM Tivoli Network Manager Reference* for information on NCIM topology databases.

The constructor and methods are described in reference (man) page format.

NCP::Domain module synopsis

The NCP::Domain module synopsis shows how to make calls to the constructor and domain operation methods that this module provides.

The comments provided in the synopsis serve as a quick reference as to the purpose of the constructor and domain operation methods. The reference (man) pages for the constructor and each method provide the details.

```
# Declare the module with the use directive.

use NCP::Domain;

# Call the NCP::Domain constructor. This call to the NCP::Domain
# constructor specifies the name of the Network Manager domain with
# the string "NEWDOMAIN". The NCP::Domain constructor returns
# to $domain a new NCP::Domain object.
#
# This call to the NCP::Domain constructor does not specify any
# database connection options.

my $domain = new NCP::Domain("NEWDOMAIN");

# Use the NCP::Domain object ($domain->) to directly invoke the
# create method to create an entry in the domainMgr table for the
# NEWDOMAIN domain.

$domain->create();

# Call the NCP::Domain constructor a second time. This call to
# the NCP::Domain constructor specifies the name of the Network
# Manager domain with a hash keyword/value pair of domain => "COPY".
# The NCP::Domain constructor returns to $copy a new NCP::Domain object.
#
# This call to the NCP::Domain constructor does not specify any
# database connection options.

my $copy = new NCP::Domain(domain => "COPY");

# Use the NCP::Domain object ($copy->) to directly invoke the clone
# method to create a new domain that is a copy of the existing
# domain (NEWDOMAIN).

$copy->clone("NEWDOMAIN");
```

```
# Call the NCP::Domain constructor a third time. This call to
# the NCP::Domain constructor specifies the name of the Network
# Manager domain with the string "OLD". The NCP::Domain constructor
# returns to $obsolete a new NCP::Domain object.

my $obsolete = new NCP::Domain("OLD");

# Use the NCP::Domain object ($obsolete->) to directly invoke
# the drop method to remove all references to the specified
# domain (OLD) from the domainMgr table.

$obsolete->drop();

# Use the NCP::Domain object ($domain->) created in the first
# call to the  NCP::Domain constructor to directly invoke the id
# method to retrieve the domain manager ID for this domain.

$domain->id();

# Use the NCP::Domain object ($domain->) created in the first
# call to the NCP::Domain constructor to directly invoke the name
# method to retrieve the domain name for this domain.

$domain->name();

# get a summary of the domain contents.

%summary = $domain->summary();
```

See Also

- “NCP::Domain Constructor” on page 277
- “clone” on page 278
- “create” on page 279
- “drop” on page 281
- “id” on page 282
- “name” on page 283
- “setLogHandle” on page 284
- “setLogLevel” on page 285

NCP::Domain Constructor

The `NCP::Domain` constructor creates a blessed `NCP::Domain` object for the specified Network Manager domain.

Constructor

```
new NCP::Domain($domainName)

new NCP::Domain($domainName, %dbOptionsHash)

new NCP::Domain(%dbOptionsHash)
```

Parameters

$domainName
Specifies the name of the Network Manager domain for which you want to create a blessed domain instance object. In this case, the domain name is specified with plain text or plain text assigned to a variable. You can also specify the domain name using the explicit hash key `"domain"`.

%dbOptionsHash
Specifies the hash that contains the database login options. One of these database login options is the domain name. More specifically, this hash takes the same database login options as the `DBI_Factory::createDbHandle` method.

Description

The NCP::Domain constructor creates a blessed NCP::Domain object for the specified Network Manager domain. Use the NCP::Domain object (for example, $domain->) to invoke the methods that the NCP::Domain module provides.

The NCP::Domain constructor provides a great deal of flexibility on how you obtain the database login options for the *%dbOptionsHash* parameter. For example, you can call the NCP::DBI_Factory::extractCmdLineOptions method to ensure that the database login options specified on the command line are provided in a common format. The return from the NCP::DBI_Factory::extractCmdLineOptions method (a reference to a hash that contains the extracted database login options and values in key/value format) is passed to the *%dbOptionsHash* parameter.

Notes

Connection to the NCIM topology database that resides in this domain will be attempted only when required.

To ensure that the NCP::Domain constructor can print appropriate messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the constructor sends these messages to STDOUT.

Example Usage

The following code fragment illustrates a typical call to the NCP::Domain constructor:

```
my $domain = new NCP::Domain("NEWDOMAIN");
```

Returns

Upon completion, the NCP::Domain constructor returns a new NCP::Domain object.

See Also

- “createDbHandle” on page 255
- “extractCmdLineOptions” on page 261
- “setLogHandle” on page 284

clone

The clone method creates a new domain that is a copy of an existing domain.

Method Synopsis

```
clone($domain)
```

Parameters

$domain
Specifies the name of an existing Network Manager domain for which you want to create a copy. You created an instance of this domain by calling the NCP::Domain constructor.

Description

The clone method creates a new domain that is a copy of an existing domain.

Notes

The `clone` method assumes that the $NCHOME environment variable is set. Otherwise, the clone method will not be able to copy the domain-specific configuration files from the existing domain.

To ensure that the `clone` method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The following code example illustrates a typical call to the `clone` method:

```
my $copy = new NCP::Domain(domain => "COPY"); 1
$copy->clone("NEWDOMAIN"); 2
```

1. This call to the `NCP::Domain` constructor does not specify any database connection options.
2. Uses the `NCP::Domain` object (`$copy->`) to directly invoke the `clone` method to create a new domain (`NEWDOMAIN`) that is a copy of the existing domain.

Returns

Upon completion, the `clone` method does not return any data.

See Also

- “NCP::Domain Constructor” on page 277
- “setLogHandle” on page 284

create

The `create` method creates an entry in the domainMgr table for the specified domain if one does not already exist.

Method Synopsis

```
create()
```

Parameters

None

Description

The `create` method creates an entry in the domainMgr table for the specified domain if one does not already exist. In addition, the `create` method creates an entry in the domainSummary table for the specified domain if one does not already exist. The domain name was specified in a previous call to the `NCP::Domain` constructor.

The domainMgr table stores data on network domains. For the specified domain, the `create` method inserts a row in the domainMgr table with values for the following table columns:

- domainName
- creationTime
- lastUpdated
- managerName
- webtopDataSource

- domainHost
- domainPort
- description

The domainMgr table contains an auto-increment column called domainMgrId, which the `create` method uses to automatically increment the field.

The domainSummary table stores statistical data on the specified domain. For the specified domain, the `create` method inserts a row in the domainSummary table with values for the following table columns:

- domainMgrId
- createTime
- changeTime

The `create` method logs appropriate error messages to a log file or STDOUT if it fails to insert an entry in the domainMgr table or the domainSummary table for the specified domain.

The `create` method permanently commits the insert row operations in the domainMgr and domainSummary tables to the database. Thus, it is not necessary for the application to call the Perl DBI module `commit` method.

Notes

To ensure that the `create` method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the `setLogHandle` method. Otherwise, the method sends these messages to STDOUT.

Network Manager applications often use the `NCP::Domain` module methods in conjunction with the methods that the `NCP::DBI_Factory` module provides.

Example Usage

The following code shows a call to the `NCP::Domain` constructor, which returns the newly created `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor. The `%$ncimArgs` hash reference contains the command line arguments (database login options and values) in key/value format returned in a previous call to the `NCP::DBI_Factory::extractcmdLineOptions` method.

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

The following code shows that the `create` method is invoked on the `NCP::Domain` object (`$domain->`). The `create` method creates entries in the domainMgr and domainSummary tables for the domain specified in *$domainName*:

```
.
.
.
$domain->create();
.
.
.
```

Returns

Upon completion, the `create` method does not return any data.

See Also

- “NCP::Domain Constructor” on page 277
- “setLogHandle” on page 284
- “NCP::DBI_Factory module reference” on page 253

drop

The drop method removes all references to the specified domain from the domainMgr table.

Method Synopsis

```
drop($domain)
```

Parameters

$domain
Specifies the name of an existing Network Manager domain for which you want to delete its associated entry from the domainMgr table. You created an instance of this domain by calling the NCP::Domain constructor.

Description

The drop method removes all references to the specified domain from the domainMgr table. This action effectively deletes any entities in the NCIM topology database for the specified domain. The drop method does not remove any configuration files associated with the specified domain.

Notes

To ensure that the drop method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Network Manager applications often use the NCP::Domain module methods in conjunction with the methods that the NCP::DBI_Factory module provides.

Example Usage

The example that illustrates a call to the drop method is divided into the following sections:

- Create a new NCP::Domain object
- Call drop to remove all references to the specified domain

Create a new NCP::Domain object

The following code shows a call to the NCP::Domain constructor, which returns a new NCP::Domain object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the NCP::Domain constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call drop to remove all references to the specified domain

The following code shows an invocation of the drop method on the NCP::Domain object ($domain->):

```
.
.
.
$domain->drop(%$scriptOptions);
.
.
.
```

Returns

Upon completion, the drop method does not return any data.

See Also

- "NCP::Domain Constructor" on page 277
- "setLogHandle" on page 284
- "NCP::DBI_Factory module reference" on page 253

id

The id method retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in the specified domain.

Method Synopsis

```
id()
```

Parameters

None

Description

The id method retrieves the domainMgrId from the domainMgr table in the NCIM topology database that resides in this domain.

Notes

To ensure that the id method can print appropriate error and other messages to a log file, you must have previously specified a log handle (that is, a reference to a file object) by calling the setLogHandle method. Otherwise, the method sends these messages to STDOUT.

Example Usage

The example that illustrates a call to the id method is divided into the following sections:

- Create a new NCP::Domain object
- Call id to return the domainMgrId

Create a new NCP::Domain object

The following code shows a call to the NCP::Domain constructor, which returns a new NCP::Domain object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the NCP::Domain constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
```

```
.
.
```

Call id to return the name of the domain

The following code shows an invocation of the id method on the NCP::Domain object ($domain->). The id method returns the domainMgrId to the *$domainMgrId* variable. Note that *$domainMgrId* is used in the call to the dropPollPolicies method.

```
.
.
.
my $domainMgrId = $domain->id();
.
.
.
dropPollPolicies($ncmonitor, $domainMgrId);
.
.
.
```

Returns

Upon completion, the id method returns the domainMgrId.

See Also

- "NCP::Domain Constructor" on page 277
- "setLogHandle" on page 284

name

The name method returns the name of the domain.

Method Synopsis

```
name()
```

Parameters

None

Description

The name method returns the name of the domain.

Example Usage

The example that illustrates a call to the name method is divided into the following sections:

- Create a new NCP::Domain object
- Call name to return the name of the domain

Create a new NCP::Domain object

The following code shows a call to the NCP::Domain constructor, which returns a new NCP::Domain object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the NCP::Domain constructor:

```
.
.
.
```

```
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call name to return the name of the domain

The following code shows an invocation of the name method on the NCP::Domain object ($domain->). The name method returns the name of the domain to the *$domainName* variable. Note that *$domainName* is used in the call to the createDbHandle method.

```
.
.
.
my $domainName = $domain->name();
.
.
.
    my $ncmonitor = NCP::DBI_Factory::createDbHandle(domain => $domainName,
                                                     dbid => "NCMONITOR",
                                                     %$ncmonitorArgs);
.
.
.
```

Returns

Upon completion, the name method returns the name of the domain.

See Also

- “NCP::Domain Constructor” on page 277
- “createDbHandle” on page 255

setLogHandle

The setLogHandle method passes in a log handle associated with an opened file to the NCP::Domain module methods. These methods can then write messages to this opened file.

Method Synopsis

```
setLogHandle($filehandle)
```

Parameters

$filehandle
Specifies a reference to a file handle (for example, IO::File) that points to an opened file to which messages can be written.

Description

The setLogHandle method passes in the log handle specified in the *$filehandle* parameter to an internal utility method called by the NCP::Domain module methods. This handle is associated with an opened file to which this internal utility method writes messages. In effect, this opened file serves as a log file that contains critical, informational, and warning type messages associated with the execution of the NCP::Domain module methods.

If you do not call the setLogHandle method, the internal utility method writes these messages to STDOUT.

Example Usage

The example that illustrates a call to the `setLogHandle` method is divided into the following sections:

- Create a new `NCP::Domain` object
- Call `setLogHandle` to log messages to a specified open file

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `setLogHandle` to log messages to a specified open file

The following code shows an invocation of the `setLogHandle` method on the `NCP::Domain` object (`$domain->`). The call to the `setLogHandle` method ensures that any messages get logged to an open file (whose associated file handle is specified in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkDomain.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    $domain->setLogHandle($logFile);
.
.
.
```

Returns

Upon completion, the `setLogHandle` method returns no data.

See Also

- “NCP::Domain Constructor” on page 277

setLogLevel

The `setLogLevel` method sets the log level for error and message reporting.

Method Synopsis

```
setLogLevel($loglevel)
```

Parameters

$loglevel

Specifies the log level to set. The following are the valid options described in ascending order:

- `debug` — Specifies a log level in which all messages are logged.
- `info` — Specifies a log level in which informational, warning, and critical messages are logged.
- `warn` — Specifies a log level in which warning and critical messages are logged.
- `critical` — Specifies a log level in which only critical messages are logged.

Description

The `setLogLevel` method sets the log level to the option specified in the *$loglevel* parameter. The default is `debug` level. If set to a higher level, only messages with an equal or higher level will be logged. For example, at level `warn`, messages of level `info` and level `debug` will not be logged.

By default, the `NCP::Domain` module methods log messages to STDOUT. If you specify a log handle to the `setLogHandle` method, the `NCP::Domain` module methods log messages to the opened file associated with this log handle.

The `setLogLevel` method logs an appropriate message (either to STDOUT or to an opened file) if you specify an invalid log level.

Example Usage

The example that illustrates a call to the `setLogLevel` method is divided into the following sections:

- Create a new `NCP::Domain` object
- Call `setLogHandle` to log messages to a specified open file
- Call `setLogLevel` to set the log level

Create a new NCP::Domain object

The following code shows a call to the `NCP::Domain` constructor, which returns a new `NCP::Domain` object to the *$domain* variable. The name of the domain is specified in *$domainName* in the call to the `NCP::Domain` constructor:

```
.
.
.
my $domain = new NCP::Domain($domainName, %$ncimArgs);
.
.
.
```

Call `setLogHandle` to log messages to a specified open file

The following code shows an invocation of the `setLogHandle` method on the `NCP::Domain` object (`$domain->`). The call to the `setLogHandle` method ensures that any messages get logged to an open file (stored in the *$logFile* local variable) rather than to STDOUT:

```
.
.
.
my $logName = "$logdir/checkDomain.$domainName.log";

    my $logFile = new IO::File;
    $logFile->open(">$logName") or die "Could not open log file $logName\n";
    $domain->setLogHandle($logFile);
.
.
.
```

Call `setLogLevel` to set the log level

The following code shows that the `setLogLevel` method is invoked on the `NCP::Domain` object (`$domain->`). The call to `setLogLevel` specifies the logging of messages at the `warn` and `critical` levels.

```
.
.
.
$domain->setLogLevel("warn");
.
.
.
```

Returns

Upon completion, the `setLogLevel` method returns no data.

See Also

- “NCP::Domain Constructor” on page 277
- “setLogHandle” on page 284

summary

The `summary` method provides a summary of the current domain.

Method Synopsis

```
%summary = $domain->summary();
```

Parameters

None.

Description

The `summary` method provides a summary of the contents of the current domain.

Example Usage

```
my %summary = $domain->summary();
```

Returns

Upon completion, the `summary` method returns a hash containing the keys and values.

```
entityData : Number of entities within this domain.
chassis    : Number of chassis entities within this domain
interface  : Number of interface entities within this domain
contains   : Number of containing entities within this domain
connects   : Number of connects within this domain
```

See Also

- “NCP::Domain Constructor” on page 277

Part 3. Database reference

Use this information to understand the structure of the Network Manager process databases and of the NCIM topology database.

Chapter 12. Discovery databases

There are various specialized databases that are used by ncp_disco, the component that discovers network device existence and connectivity, and by ncp_model, the component that manages, stores, and distributes the discovered network topology.

The ncp_disco component and ncp_model component store configuration, management, and operational information in databases. You can interrogate these databases by logging into the DISCO or MODEL service using the OQL Service Provider.

The ncp_disco databases can either be active or passive. When data is inserted into an active database, an action is automatically triggered; for example, another table is populated with data, or a script or stitcher is launched.

Discovery engine database

Use the Discovery engine (ncp_disco) database to configure the general options for the discovery process, and to track the discovery process.

The Discovery engine database, `disco`, is defined in `$NCHOME/etc/precision/DiscoSchema.cfg`.

disco.agents table

The agents table specifies the discovery agents that **ncp_disco** uses for the discovery. Every agent that you want to run must have an insertion into the disco.agents table within the `DiscoAgents.cfg` configuration file that enables that agent (set m_Valid=1). If m_Valid=0, the agent is not run.

Table 75. disco.agents database table schema

Column name	Constraints	Data type	Description
`m_AgentClass`		Integer	The category to which the current discovery agent belongs: • (0) Routing agent • (1) Switch agent • (2) Hub agent • (3) ILMI agent • (4) FDDI agent • (5) PNNI agent • (6) Frame Relay agent • (7) CDP agent • (8) NAT agent
`m_AgentName`	• PRIMARY KEY • NOT NULL	Text	The unique name of the discovery agent.
`m_DebugLevel`		Integer	The level of debugging for the agent.
`m_EndSignal`			Defines what signal to use to stop the agent. This property can be used if directed by IBM Support, but is not defined in the `DiscoSchema.cfg` table.

Table 75. disco.agents database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HostName`		Text	The name of the host machine on which to run the agent.
`m_IsIndirect`		Integer	A flag indicating the type of connectivity information returned by the discovery agent: • (0) Direct connectivity; for example, Routing agents • (1) Indirect connectivity information; for example, Switch agents
`m_LogFile`		Text	The text file to which debugging output is written.
`m_Precedence`		Integer	An integer representation of the precedence level of the information returned by the discovery agent; the higher the integer, the higher the weighting given to the information returned. The precedence is only used when there is a conflict when merging device information to produce the workingEntities.finalEntity database table.
`m_MessageLevel`		Text	Specifies the message level (the default is warn). Options include: • debug • info • warn • error • fatal
`m_NumThreads`		Integer	The number of threads this agent runs. If not specified, the default number is 10; the maximum allowed is 900.
`m_Valid`		Integer	A flag determining whether or not the discovery agent is to be used: • (1) Run the discovery agent • (0) Do not run the discovery agent
`m_ValidOnPartial`		Integer	Specifies whether the agent is to be used on a partial discovery: • 0: Agent is *not* to be used in a partial discovery. • 1: Agent is to be used in a partial discovery.

The disco.agents table also indicates agent precedence, which can be used when merging device information to produce the workingEntities.finalEntity table. Precedence determines which records are used when duplicate or conflicting records are reported by different discovery agents.

The following precedence applies:

- The Details agent has the lowest precedence because it is designed to retrieve only basic device information.
- Routing agents have the next highest precedence. Their connectivity information is at the IP layer only, however, so it is not as accurate as that returned by the switching agents.
- Switching agents have next-highest precedence because they can return information on the media layer (layer 2), which is more accurate than layer 3 information.

disco.config table

The config table configures the general operation of the discovery process.

Table 76. disco.config database table schema

Column name	Constraints	Data type	Description
m_AddIntDisplayLabel		Boolean Integer	Specifies whether to add an interface display label: • 0: Do not add an interface display label. • 1: Add an interface display label.
m_AllowVirtual	Default = 1	Integer	Flag indicating whether to allow virtual IP addresses as part of the discovery. • 0: Do not perform any discovery for virtual IP addresses. • 1: Perform discovery for virtual IP addresses. This is the default setting. • 2: Perform discovery for virtual IP addresses only if the address is defined in the scope.special table. This table defines management IP addresses.
m_BuildLogicalCollections		Boolean Integer	Specifies whether to build logical collection entities to group together items such as VTP domains, OSPF areas, and MPLS VPNs: • 0: Do not build logical collection entities. • 1: Build logical collection entities.
m_CheckFileFinderReturns	Externally defined Boolean data type default = 0	Boolean Integer	Flag indicating whether to use the Ping finder to check the devices specified in the flat file supplied to the File finder. • 1: This setting tells the Ping finder to check the devices specified in the flat file supplied to the File finder. This setting is recommended if you have reason to doubt that some of the devices specified in the flat file are still connected to the network. • 0: This setting indicates that you do not want to perform any checking of the devices specified in the flat file supplied to the File finder.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_CycleLimit		Integer	The number of discovery cycles to complete before instigating a full rediscovery (used by the FinalPhase stitcher).
m_CreateStchrEvents	Externally defined Boolean data type default = 1	Boolean Integer	Specifies whether to create discovery events to be sent to the ObjectServer. This field takes the following values: • 0: Do not generate discovery events. • 1: Generate discovery events.
m_CustomIntNaming		Boolean Integer	Changes the default naming convention for discovered interfaces. If you change the default naming convention for discovered interfaces, you must change the BuildInterfaceName stitcher to specify your naming convention.
m_DirScanIntvl		Integer	The time period between scans for modifications to the stitcher and agent files. If changes are found, the stitcher and agent definitions are loaded and the appropriate changes are made to the stitchers and agents.
m_DiscoProfiling		Boolean Integer	Flag indicating whether to profile the discovery. • 1: Profile the discovery. • 0: Do not profile the discovery.
m_DiscoOnStartup		Boolean Integer	Specifies whether a discovery should automatically start when the Discovery engine, ncp_disco, is started: • 0: Do not automatically start a discovery. • 1: Automatically start a discovery.
m_DisplayMode		Integer	Specifies how the display label used for GUI network and network hop views should be populated for main nodes. • 0 - Use Entity Name (default) • 1 - Use SysName. This option is useful when it is not desirable to name entities by sysName in the database (see m_UseSysName) but it is desirable to have the entities displayed in the GUI views with a sysName.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_FeedbackCtrl	Default = 0	Integer	Flag indicating whether to use the feedback mechanism during the discovery. The feedback mechanism allows any new IP addresses to be fed back into the discovery and thus increases the size of the discovered network. Devices that are fed back are pinged by the Ping finder. **Note:** For feedback to work the ping finder must be activated. • 0: Feedback is off for all discoveries and rediscoveries. This option provides speed but discovers only those devices specified to the finders, and hence provides an incomplete topology. However, this setting ensures that discoveries and rediscoveries complete in the quickest possible time. • 1: Feedback is on for full discoveries, full rediscoveries, and partial rediscoveries. All IP addresses are pinged. This option provides a complete topology in all situations but takes the longest. • 2: Feedback is on for full discoveries and full rediscoveries, this ensuring a complete topology in these cases. In the case of partial rediscoveries there is no feedback. This ensures that the partial rediscovery runs in the quickest time possible. This is the default setting.
m_FindersOnStartup		Boolean Integer	Specifies whether the finders should automatically start when the Discovery engine, ncp_disco, is started : • 0: Do not automatically start finders. • 1: Automatically start finders.
m_EnableCrossDomainProcessing		Boolean Integer	Set to 1 to enable cross-domain processing. If you are performing a cross-domain discovery, you must perform other configuration steps in addition.
m_InferCEs	Externally defined Boolean data type default = 0	Boolean Integer	Flag indicating whether to infer the existence of customer-edge (CE) routers. When enabled, DISCO creates a CE router entity for each provider-edge (PE) router interface that is on a /30 subnet and does not have CE information from another source. • 1: This setting tells DISCO to infer the existence of CE routers. • 0: This setting tells DISCO not to infer the existence of CE routers.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_InferPEsUsingBGP		Boolean Integer	Specifies whether to infer the existence of provider-edge (PE) routers using BGP information on customer-edge (CE) routers: • 0: Do not infer PEs. • 1: Infer PEs.
m_ManagedWaitTimeOut		Integer	Applicable when the value of the field `m_WaitForManagedProcs` is set to 1. Maximum time to wait for all collectors to complete retrieving data from their EMSs. Effectively a fail-safe mechanism to cover the siutation where one of the collectors never completes processing. By default, this value is set to 0, which means wait indefinitely.
m_MinResidentSize		Integer	The minimum initial size of DISCO in kilobytes (KB). The maximum value that you can specify is 500 MB (512 000 KB). Specifying an initial value speeds up the discovery by allocating the memory of DISCO in one block.
m_ModelVlans	Externally defined Boolean data type	Boolean Integer	Flag indicating whether to switch off VLAN modelling. 1: This setting switches *on* VLAN modelling. When you make this setting, the AddGlobalVlans, CreateTrunkConnections and AddVlanContainers stitchers *are* called. 0: This setting switches *off* VLAN modelling. When you make this setting, the AddGlobalVlans, CreateTrunkConnections and AddVlanContainers stitchers are *not* called.
m_NothngFndPeriod		Float	The maximum time lapse, in seconds, between the discovery of one device and the discovery of the next device in the device discovery phase.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_OspfPidNaming	Default=0	Integer	Fix Pack 2 Defines whether OSPF process IDs (PIDs) are used when naming and creating OSPF-related entities. Leave this setting at the default (off) unless your network has a meaningful OSFP process ID allocation policy and you are running discovery agents capable of discovering process IDs from the target devices. The possible options are the following: **0 - (default)** Do not use PIDs when naming OSPF entities. **1 - Name OSPF Areas by PID** Use PID and area ID when creating and naming area entities. Areas collect interfaces with the same domain, area and PID. **2 - Name OSPF domains and Areas by PID** Use PID and area ID when creating and naming area entities. Areas collect interfaces with the same domain, area and PID. Create per-process ID OSPF routing domains in addition to the standard PID-agnostic domains. Per-PID domains collect only areas with the required PID.
m_PendingPerCent		Integer	The maximum allowed ratio of pending devices to processing devices. A breach of this threshold condition instigates a full discovery (rather than a partial rediscovery).
m_PingVerification	Default = 2	Integer	Option to check whether an interface is able to be pinged. If the device is not pingable, then Network Manager does not poll the device for alerts • 0: Do not check pingability: Network Manager performs no pingability check on any of the interfaces discovered. Interfaces will be polled regardless of whether they are pingable at discovery time. • 1: Check pingability: perform a pingability check, following discovery, for every interface discovered. • 2: Determine best method: sets the pollability flag for an interface based on whether feedback was active during the discovery..

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_RebuildLayers	Externally defined Boolean data type	Boolean Integer	Flag indicating whether to rebuild topology layers after partial rediscovery. • 1: Rebuild the layers. After partial rediscovery, topology layers stitchers are run. Partial rediscovery takes longer but results in a complete topology. • 0: Do not rebuild the layers. After partial rediscovery, topology layers stitchers are not run. The result is a much quicker partial discovery; however, connectivity associated with the newly discovered device is not fully seen in the topology.
m_RediscoverRelatedDevices		Boolean Integer	In a partial rediscovery when a device has changed, specifies whether to rediscover the related devices if the connection appears to have changed: • 0: Do not rediscover the related devices if the connection appears to have changed. • 1: Rediscover the related devices if the connection appears to have changed.
m_RefreshDiscovery	Default = 1	Boolean Integer	Specifies whether the FullDiscovery stitcher restarts the discovery when called after an initial full discovery has completed. The default value is 1: restart the discovery process. Set the value to 0 to not restart the discovery process using the RestartDiscoProcess.stch stitcher. Enabling this option by default is useful for the following reasons: • If the discovery is loading custom data into the `DiscoContrib.cfg` file then this option facilitates the process by having the new discovery process read the file again. • The option also helps if the discovery process is accumulating memory because the newly started process resets the process to the initial state **Note:** The FullDiscovery stitcher only stops and starts the discovery process if there is no discovery in progress at the time it is called.
m_RestartAgents		Integer	A flag that determines whether DISCO attempts to restart discovery agents that fail during their operation.
m_RestartFinders		Integer	Flag to determine whether to restart a failed finder.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_RTVPNResolution		Integer	Specifies whether to apply fine control over Layer 3 VPN resolution and naming in a route target-based discovery: • 1: Use route target. • 2: Use VRF (default).
m_SubnetFiltering		Integer	Alters which interfaces are included in subnet-based connections: • 0: No filtering • 1: Filter out VRF interfaces (consider using m_VpnASTagging instead of this mode as it improves connectivity in all layers rather than just layer 3). • 2 - Filter out in-scope interfaces known to hold inaccessible duplicate IPs. • 3 - Automatic. Decides best approach based on other configuration options.
m_UseIfIndex		Boolean Integer	Specifies whether to name interfaces using the ifIndex only. This setting overrides the m_UseIfName setting. • 0: Do not name interfaces using the ifIndex only. • 1: Name interfaces using the ifIndex only.
m_UseContext		Boolean Integer	Flag indicating whether this is a context-sensitive discovery. • 1: Specifies a context-sensitive discovery. • 0: Specifies that this is not a context-sensitive discovery.
m_UseIPName	Externally defined Boolean data type default = 0	Boolean Integer	Flag to indicate which naming strategy to use for devices. • 1: Uses the IP address for device naming. • 0 (default): Does not use IP address for device naming. Instead, the DNS name is used where available.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_UseIfName	Externally defined Boolean data type default = 0	Boolean Integer	Flag indicating which naming strategy to use when building interfaces. • 1: This setting indicates that you want to use ifName or ifDescr to name the interfaces rather than their ifIndex, card or port information. • 0: This setting indicates that you want to use the default naming convention for any device interface: `baseName[<card>[<port>]`
m_UseSysName	Externally defined Boolean data type default = 0	Boolean Integer	Flag indicating which naming strategy to use when naming devices. • 1: This setting indicates that you want to name devices using the value of the SNMP sysName variable as the main source of naming information. The sysName variable must be set and must be unique within the network. • 0: This setting indicates that you do not want to name devices using the value of the SNMP sysName variable as the main source of naming information.
m_VerifyCDPUsingDeviceId		Boolean Integer	Specifies whether to verify the CDP links using the CDP device ID. Occasionally the CDP device ID has been found to be unreliable. Switching on `m_VerifyCDPUsingDeviceId` will improve CDP connectivity if the Device ID is accurate but might degrade connectivity if the Device ID is inaccurate. • 0: Do not verify the CDP links using the CDP device ID. • 1: Verify the CDP links using the CDP device ID.
m_VpnASTagging		Integer	Specifies whether CE facing PE interfaces should be assigned to a private address space: • 0: Do not assign. • 1: Assign. • 2: Automatic. Decides the best approach based on other configuration options.

Table 76. disco.config database table schema (continued)

Column name	Constraints	Data type	Description
m_WaitForManagedProcs		Boolean Integer	Flag indicating whether to tell the discovery engine ncp_disco wait until the Collector finder status is complete before moving to the next phase of discovery. • 1: Wait until the Collector finder status is complete before moving to the next phase of discovery. Setting m_WaitForManagedProcs = 1 when running a multiple collector discovery forces the discovery process to remain in phase 1 until all the collectors have completed data processing on their respective EMSs. • 0: Do not wait until the Collector finder status is complete before moving to the next phase of discovery. When this field is set to 1, then the parameter `m_ManagedWaitTimeOut` defines the maximum time to wait for all collectors to complete retrieving data from their EMSs.
m_WriteTablesToCache	Externally defined Boolean data type	Boolean Integer	Flag indicating whether to write a cache of the Discovery engine, ncp_disco, tables to disk. **Note:** Setting this flag results in discoveries that are slower than a standard discovery. • 1: Write cache of ncp_disco tables to disk. The tables that are defined in the failover database are cached and ncp_disco can be restarted at any point. • 0: Do not write cache of ncp_disco tables to disk. No tables are cached during the discovery and ncp_disco ignores any existing cache files if it is restarted.
m_UnmanagedSubInts		Boolean Integer	Flag indicating whether to automatically set sub-interfaces to unmanaged when the parent interface is marked as unmanaged by the PopulateDNCIM_ManagedStatus.stch stitcher. • 1: Automatically set sub-interfaces to unmanaged if the parent interface is set to unmanaged by the stitcher. • 0: Do not automatically set sub-interfaces to unmanaged.

disco.convergedTopologies table

The convergedTopologies table stores information on how layers must be merged by the discovery process.

Table 77. disco.convergedTopologies database table schema

Column name	Constraints	Data type	Description
m_TopologyName	• PRIMARY KEY • NOT NULL	Text	Name of the topology to be created from the source topologies.
m_Order		Integer	Order in which to process this entry.
m_SourceTopologies		List of text	Topologies to be used in the merge, by order of most accurate topology first.
m_RetainOverride		Boolean Integer	Controls whether the override flags found in the source topologies are to be put into the merge topology. This value is only relevant when the merged topology is going to be used as a source topology in another merged topology. • 0: Do not copy override flags. • 1: Copy override flags.
m_UseContainment		Boolean Integer	Provides fine control over whether interface containment information is taken into consideration when merging this particular topology. • 0: Do not try to use containment to refine links. • 1: Allow links on interfaces higher in the containment to be overridden by any links on interfaces lower down; that is, more physical interfaces.

disco.dynamicConfigFiles table

The dynamicConfigFiles table stores the names of configuration files that must be reread each time a full discovery is launched.

Table 78. disco.dynamicConfigFiles database table schema

Column name	Constraints	Data type	Description
`m_Name`	Primary key Not null	`Text`	Name of configuration file to be reread,
`m_UpdTime`		`Timestamp`	Last update time for this configuration file.

disco.events table

The events table constrains discovery events generated to a standard format. An event is generated by inserting a record into this table.

Table 79. disco.events database table schema

Column name	Constraints	Data type	Description
`m_EventName`	Not null	`Text`	The name of the event.
`m_EntityName`	Not null	`Text`	The name of the entity on which the event occurred.
`m_EventType`	Not null	`Integer`	This field can take one of the following values: • 1: Problem • 2: Resolution • 13: Informational
`m_Severity`	Not null	`Integer`	This field can take one of the following values: • 0: CLEAR • 1: INDETERMINATE • 2: WARNING • 3: MINOR • 4: MAJOR • 5: CRITICAL It is possible to define more values.
`m_Description`	Not null	`Text`	Description of the discovery event
`m_ExtraInfo`	Externally defined vblist data type		Specifies a list of additional information.

disco.filterCustomTags table

The filterCustomTags table stores custom tags, which can be associated with a filtered set of discovered entities during the discovery and used to perform custom visualization and polling tasks.

Table 80. disco.filterCustomTags database table schema

Column name	Constraints	Data type	Description
m_Filter		Text	Filter definition that extracts a set of IP address to which the name-value pair tags in the m_CustomTags is to be associated to. You can create a filter based on any attributes associated with discovered entities. For example, you could apply the following filters: • Filter based on entity IP address: "m_UniqueAddress LIKE '172.20.3'" • Filter based on entity name: "m_Name LIKE 'lon'" • Filter based on VLAN identifier of a VLAN entity: "m_LocalNbr->m_VlanID = 102"
m_StitcherTagName		Text	Name of a tag to be evaluated using the GetTagStitcher.stch stitcher.
m_CustomTags		Object type vblist	List of name-value pair tags.

disco.ipCustomTags table

The ipCustomTags table stores custom tags, which can be associated with unique discovered entities during the discovery and used to perform custom visualization and polling tasks.

Table 81. disco.ipCustomTags database table schema

Column name	Constraints	Data type	Description
m_UniqueAddress		Text	IP address to which the name-value pair tags in the m_CustomTags is to be associated to.
m_StitcherTagName		Text	Name of a tag to be evaluated using the GetTagStitcher.stch stitcher.
m_CustomTags		Object type vblist	List of name-value pair tags.

disco.managedProcesses table

The managedProcesses table is a repository for all the subprocesses managed by DISCO, such as the finders. Provided that CTRL is running, processes that are inserted into this table are started and managed by DISCO.

Table 82. disco.managedProcesses database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • UNIQUE • NOT NULL	Text	The name of the process to be managed.
`m_Args`		List of text	A list of command-line arguments sent to the executable.
`m_Host`		Text	The name of the host on which to run the executable.
`m_LogFile`		Text	The name of the log file to which output is written.

disco.NATStatus table

The NATStatus table is used to configure the discovery system to use NAT.

Table 83. disco.NATStatus database table schema

Column name	Constraints	Data type	Description
m_NATChecks		Integer	A counter for unresponsive NAT Gateways that were configured for discovery. **Important:** Do not change the value of this field unless you are advised to do so by IBM Software Support.
m_NATStatus	• UNIQUE • NOT NULL	Integer	This column is populated automatically by the discovery process, and can be used to track the process of a NAT discovery. To make inserts into this table, set the value to `0`. Possible values are as follows: • `0`: Uninitialized • `1`: Seeded discovery with gateways • `2`: Awaiting gateway returns • `3`: Processing NAT translations • `4`: NAT translations complete
m_UsingNAT	• UNIQUE • NOT NULL	Boolean integer	Indicates whether the discovery uses multiple address spaces. If the discovery uses multiple address spaces, set the value to `1`. If not, set the value to `0`.

disco.profilingData table

The profilingData table is used by the discovery profiling stitchers to record data associated with time and memory expended during the discovery.

Table 84. disco.profilingData database table schema

Column name	Constraints	Data type	Description
m_AgentData		List	For each agent that ran during the discovery, provides a brief summary of the work performed by that agent during the discovery.
m_CompletionMem		64-character string	Memory used when phase -1 of the discovery completed.
m_CompletionTime		Integer	Time that phase -1 completed.
m_DiscoveryMode		Integer	Type of discovery: • 0: Full discovery • 1: Partial discovery
m_NumDetailsReturns		Integer	Total number of details table returns during the discovery.
m_NumEntities		Integer	Total number of entities in the dNCIM database based on the count of NCIM topology database entityData table records for the domain.
m_NumFinderInserts		Integer	Total number of finder inserts during the discovery.
m_NumMainNodes		Integer	Total number of main nodes discovered.
m_NumMainNodesWithAccess		Integer	Total number of main nodes with SNMP access discovered.
m_NumIPs		Integer	Total number of IP addresses discovered.
m_NumRouters		Integer	Total number of routing devices discovered.
m_NumSwitches		Integer	Total number of switches discovered.
m_Phase1StartMem		64-character string	Memory used when phase 1 of the discovery started.
m_Phase1StartTime		Integer	Time that phase 1 of the discovery started. Phase 1 is also known as the Interrogating Devices phase.

Table 84. disco.profilingData database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_Phase2StartMem		64-character string	Memory used when phase 2 of the discovery started.
m_Phase2StartTime		Integer	Time that phase 2 of the discovery started. Phase 2 is also known as the Resolving Addresses phase.
m_Phase3StartMem		64-character string	Memory used when phase 3 of the discovery started.
m_Phase3StartTime		Integer	Time that phase 3 of the discovery started. Phase 3 is also known as the Downloading Connections phase.
m_ProcPhaseStartTime		Integer	Time that phase -1, the data processing phase of the discovery started. Phase -1 is also known as the Correlating Connections phase.
m_ProcPhaseStartMem		64-character string	Memory used when phase -1 of the discovery started.
m_SoftwareVersion		Text	Software version used.
m_StitcherData		List	For each stitcher that ran during the discovery, provides information on how long the stitcher ran for, and how many times it was executed during the discovery. This enables the identification of problem stitchers during a discovery.

disco.status table

Use the disco.status table to monitor the progress of the **ncp_disco** process during the discovery process.

Attention: The disco.status table is used and updated internally, and you must not make inserts into this table.

Table 85. disco.status database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_BlackoutState	Externally defined Boolean data type	Boolean Integer	Flag to show if the discovery process is in Blackout mode, that is, whether or not DISCO is accepting new devices from the finders in the current discovery cycle: • 0: False (accepting new devices) • 1: True (not accepting new devices)

Table 85. disco.status database table schema (continued)

Column name	Constraints	Data type	Description
m_CycleCount		Integer	Current rediscovery cycle, that is, the current number of cycles DISCO has been through without actually building a topology. In rediscovery mode, DISCO only builds a topology at the end of the last cycle (the last cycle is determined by the fact that there is nothing left in `finders.pending` awaiting processing).
m_DiscoveryCycle Requested	Externally defined Boolean data type	Boolean Integer	Flag to indicate that a discovery has been requested by the GUI.
m_DiscoveryCycle RequestTime		Integer	The time that the discovery was requested, in Unix time.
m_DiscoveryMode		Integer	The present discovery mode: • 0: Full discovery • 1: Partial discovery
m_ForcedLayerRebuild		Boolean Integer	Flag that indicates whether a 'false' value of the disco.config.m_RebuildLayers column has been overridden for the current discovery cycle. The default value is 0.
m_FullDiscovery	Externally defined Boolean data type	Boolean Integer	Flag to indicate that the FullDiscovery.stch stitcher has been called during the discovery. If the stitcher is called, the flag is set to 1 to ensure that the FullDiscovery.stch stitcher is executed when the current discovery finishes (thus starting another full discovery). If the flag is set to any other value, no action is taken.

Table 85. disco.status database table schema (continued)

Column name	Constraints	Data type	Description
m_Phase		Integer	The current phase within the present discovery cycle. During the data collection stage, the phases are as follows: • 0: The discovery has not yet started. • 1: The main discovery phase in which device data is retrieved. Most discovery agents complete in this phase. • 2 - n: The phases in which the topology data is retrieved for the currently discovered objects. The number of phases required depends on how your discovery is configured. By default, in a layer 2 discovery, phase 2 consists of the retrieval of IP to MAC address translations and phase 3 consists of the retrieval of Ethernet switch topology information. During the data processing stage, the following phase is undertaken. • 3: The phase in which the collected data is processed; the layers are built and the data is sent to MODEL. More detailed information about the discovery phases can be found in "Discovery stages and phases" on page 797.
m_ProcessingNeeded	Externally defined Boolean data type	Boolean Integer	Flag to indicate whether the current topology needs reprocessing. This flag is checked when DISCO is in rediscovery mode in order to determine whether any newly found devices (that are now in the finders.pending table) necessitate the reprocessing of the entire topology: • 0: The topology does not need reprocessing • 1: The topology needs reprocessing

disco.tempData table

The tempData table is used by the discovery profiling stitchers to record the time and memory expended to perform the discovery.

Table 86. disco.tempData database table schema

Column name	Constraints	Data type	Description
m_Phase1TmpTime		Integer	Time taken by phase 1 of the discovery, also known as the Interrogating Devices phase.
m_Phase2TmpTime		Integer	Time taken by phase 2 of the discovery, also known as the Resolving Addresses phase.
m_Phase3TmpTime		Integer	Time taken by phase 3 of the discovery, also known as the Downloading Connections phase.
m_ProcPhaseTmpTime		Integer	Time taken by phase -1, the data processing phase of the discovery, also known as the Correlating Connections phase.
m_Phase1TmpMem		64-character string	Memory used during phase 1 of the discovery.
m_Phase2TmpMem		64-character string	Memory used during phase 2 of the discovery.
m_Phase3TmpMem		64-character string	Memory used during phase 3 of the discovery.
m_ProcPhaseTmpMem		64-character string	Memory used during phase -1 of the discovery.

Example configuration of the disco.agents table

This example uses OQL commands to insert configuration values into the disco.agents table.

- The ArpCache discovery agent is enabled to run during the discovery (m_Valid=1), belongs to the routing class (m_AgentClass=0), returns direct connectivity information (m_IsIndirect=0) and has a precedence level of 2.
- The AtmForumPnni discovery agent is disabled for this discovery (m_Valid=0), belongs to the PNNI class (m_AgentClass=5), returns direct connectivity information (m_IsIndirect=0) and has a precedence level of 5.
- The BayEthernetHub discovery agent is disabled for this discovery (m_Valid=0), belongs to the hub class (m_AgentClass=2), returns indirect connectivity information (m_IsIndirect=1) and has a precedence level of 3.

```
insert into disco.agents
(
        m_AgentName, m_Valid, m_AgentClass, m_IsIndirect, m_Precedence
)
values
(
        'ArpCache', 1, 0, 0, 2
);
```

```
insert into disco.agents
(
        m_AgentName, m_Valid, m_AgentClass, m_IsIndirect, m_Precedence
)
values
(
        'AtmForumPnni', 0, 5, 0, 5
);

insert into disco.agents

(
        m_AgentName, m_Valid, m_AgentClass, m_IsIndirect, m_Precedence
)
values
(
        'BayEthernetHub', 0, 2, 1, 3
);
```

Example configuration of the disco.config table

This example uses OQL commands to insert configuration values into the disco.config table.

- The maximum period between device discovery is 300 seconds. This condition and the next condition must be satisfied in order to proceed to the next phase of the discovery cycle.
- The maximum allowable ratio of pending to processing devices is 20 percent. If this threshold is breached, a full discovery is instigated.
- The cycle limit is 5, which means that a maximum of five discovery cycles are necessary to complete the discovery process. If there are more than 5 discovery cycles, a full rediscovery is initiated.
- The agent restart flag is 1, which means that DISCO is mandated to restart any discovery agent that fails in its operation.
- The finder restart flag is 1, which means that DISCO is mandated to restart any finder that fails in its operation.
- Scans for updates to the agents and stitchers have been disabled. This is usually the case when you do not wish to alter the discovery data flow.
- Do not write a cache of the Discovery engine, ncp_disco, tables to disk.

```
insert into disco.config
(
            m_NothngFindPeriod,
            m_PendingPerCent,
            m_CycleLimit,
            m_RestartAgents,
            m_RestartFinders,
            m_DirScanIntvl,
            m_WriteTablesToCache
)
values
(
            300,
            20,
            5,
            1,
            1,
            0,
            0
);
```

Example configuration of the disco.managedProcesses table

This example uses OQL commands to insert configuration values into the disco.managedProcesses table. If the CTRL program is running, you can configure, launch, and manage the File finder and Ping finder subprocesses.

```
insert into disco.managedProcesses
(
```

```
            m_Name, m_Args, m_Host
)
values
(
            "ncp_df_file", [ ], "othello"
);

insert into disco.managedProcesses
(
            m_Name, m_Args, m_Host
)
values
(
            "ncp_df_ping", [ ], "othello"
);
```

Discovery scope database

The scope database limits the extent or reach of a discovery. Using the scope database, you can configure a range of protocols and attributes that define zones that are to be included or excluded from the discovery process.

The range of IP addresses and devices that can potentially be considered by the discovery process is unlimited, so unless you restrict the scope of the discovery, ncp_disco would eventually attempt to discover the entire Internet.

For example, you can specify that sensitive devices not be discovered and consequently not be instantiated. A sensitive device is one that you do not want to poll. This might be because there is a security risk involved in polling the device, or that polling might overload device.

disco.scope database schema

The scope database is defined in $NCHOME/etc/precision/DiscoSchema.cfg and $NCHOME/etc/precision/DiscoScope.cfg. Its fully qualified database table names are: scope.zones; scope.detectionFilter; scope.instantiateFilter; scope.special.

scope.detectionFilter table

If you specify a filter in the detectionFilter table, only devices matching it are discovered.

Table 87. scope.detectionFilter database table schema

Column name	Constraints	Data type	Description
m_Filter		Text	A textual representation of an attribute filter against the columns of the Details.returns table; for example, m_UniqueAddress or m_ObjectId.

Although you can configure the filter condition to test any of the columns in the `Details.returns` table, when filtering IP devices you might need to use the IP address as the basis for the filter if you need to restrict the detection of a particular device. If the device does not grant SNMP access to the Details agent, the Details agent might not be able to retrieve MIB variables such as the Object ID. However, you are guaranteed the return of at least the IP address when the device is detected.

scope.inferMPLSPEs table

Use the scope.inferMPLSPEs table when enabling inference of inaccessible provider-edge (PE) devices by using the BGP data on the customer-edge (CE) devices. This table enables you to optionally specify which zones to process to determine which of the inferred PE devices are valid devices.

To specify which zones to process to determine which of the inferred PE devices are valid devices populate the scope.inferMPLSPEs table, using standard format scope entries, as in the scope.zones table.

Use this option when you have inaccessible devices that are connected by means of BGP but which are not actually PE devices.

If the following conditions are true, then the system creates a "third-party" network object to model this inaccessible provider network.

- A router is within this scope
- The router has BGP peers outside the discovered network
- m_InferMPLSPEsUsingBGP is on. This can also be defined using the **Advanced** tab on the Discovery Configuration GUI.

Table 88. scope.inferMPLSPEs database table schema

Column name	Constraints	Data type	Description
`m_Protocol`	• PRIMARY KEY • NOT NULL • Externally defined netProtocol data type	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`m_Action`	• NOT NULL • Externally defined filtAction data type	Integer	Action to perform for current zone: • 0: Undefined • 1: Include • 2: Exclude
`m_Zones`	NOT NULL	List of type zone	A list of varbinds (name=value) that define the present discovery zone.

Only process interfaces in the 199.220.* network

The following example shows how to instruct the system to only process interfaces in the 199.220.* network.

```
insert into scope.inferMPLSPEs
(
     Protocol,
     m_Action,
     m_Zones
)
values
(
     1,
     1,
     [ { m_Subnet = "199.220.*" } ]
);
```

scope.instantiateFilter table

When you specify a filter in the instantiateFilter table, only devices that pass the criteria are instantiated, that is, sent to the DNCIM database. If no filter is specified, all discovered devices are instantiated.

Note that because the m_Protocol column must be unique, there must be only one insert into this table for any given protocol. Multiple filters must be defined within a single insert.

Table 89. scope.instantiateFilter database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Filter		Text	A textual representation of an attribute filter against the columns of the ncimCache database. The following example insert prevents instantiation of an interface: `insert into scope.instantiateFilter ( m_Filter ) values ( " ( BASENAME != 'jane' OR ( BASENAME = 'jane' AND networkInterface->IFINDEX != 1 ) ) " );`

scope.mplsTe table

The scope.mplsTe table defines the scope of MPLS Traffic Engineered (TE) tunnel discovery, and defines what information is retrieved.

The following table shows the schema of the scope.mplsTe table.

Table 90. scope.zones database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Protocol	• NOT NULL • Externally defined netProtocol data type	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
m_Zones	NOT NULL	List of type zone	Defines the scope in which the tunnel heads will be discovered
m_AddressSpace		Text	Optional address space

Table 90. scope.zones database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_Mode		Integer	The TE tunnel discovery mode defines what information is retrieved. Possible values are: • 0: Unknown (not set) • 1: Tunnel Head/Tail with Transit Hop List • 2: Tunnel Head/Tail (No Hop List) • 3: Tunnel Head, Tails, and Transit devices
m_TunnelFilter		Integer	The TE tunnel filter. Possible values are: • 0: Unknown (not set) • 1: Include tunnels with this head • 2: Exclude tunnels with this head

scope.multicastGroup table

The scope.multicastGroup table defines which multicast groups to discover and which details to retrieve from these groups.

The following table shows the schema of the scope.multicastGroup table.

Table 91. scope.multicastGroup database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_AddressSpace		Text	Optional address space
m_GroupName		Text	Descriptive name for a group
m_Groups	Not null	list type zone	Zones defines the multicast subnets to which the scope options apply
m_IGMPMode		Integer	IGMP Group discovery mode • 0 - unknown (use default) • 1 - Include group • 2 - Exclude group
m_IPMRouteMode		Integer	IP Multicast Route Group discovery mode: • 0 - unknown (use default) • 1 - Include group • 2 - Exclude group

Table 91. scope.multicastGroup database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_PimMode		Integer	The PIM multicast discovery mode defines what information is retrieved. Possible values are: • 0: Unknown (use default) • 1: Retrieve PIM group data • 2: Do not retrieve PIM group data. Groups with this option applied will not be represented in the PIM Service/End Point data.
m_Protocol	• NOT NULL • Externally defined netProtocol data type (Currently IPv4 [1] only)	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6

scope.multicastSource table

The scope.multicastSource table defines which IPM routes to discover. This is particularly useful if you have multiple IPM route sources, since you can scope multicast discovery by IPM route source to focus on the sources of interest.

The following table shows the schema of the scope.multicastSource table.

Table 92. scope.multicastSource database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Protocol	• NOT NULL • Externally defined netProtocol data type	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
m_Source	NOT NULL	list type zone	The multicast source to be included or excluded

Table 92. scope.multicastSource database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_IPMRouteMode		Integer	An integer representation of the network protocol used by the presently defined group. The following values are possible: • IP Multicast Route Source discovery mode • 0 - unknown (use default) • 1 - Include Source • 2 - Exclude Source
m_Groups		list type zone	The multicast group subnets to which the source scope option applies

scope.special table

The special table defines management IP addresses. A management address is an IP address on a device whose only role is to manage the device. Management addresses do not handle network traffic.

Table 93. scope.special database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_Zones`	NOT NULL	List of type zone	A list of varbinds (name=value) that define the present discovery zone. This takes the form of a list of subnet IP addresses and subnet.
`m_AddressSpace`		Text	Optional address space identifier for a particular scope entry.
`m_Protocol`		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`m_OutOfBand`		Int Type Boolean	Indicates whether the management area is out of band. Takes one of the following values: • 0: in band • 1: out of band

Table 93. scope.special database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_IsManagement`		Int Type Boolean	Indicates whether the address is a management address.
`m_IsValidVirtual`		Int Type Boolean	Indicates whether the address is a valid virtual IP.
m_Identifier		Text	Optional identifier for tracking.
m_Priority		Int	Priority that is used if there are multiple matches. The scope.special entry that has the highest priority is selected. This priority must be set to at least 1.
m_NonPingable		Int	If set to 1, the address is selected even if it cannot be pinged.
m_AdminInterface		Int Type Boolean	Indicates whether the address is an interface.
m_ExtraInfo		Object type vblist	Optional fields with which the target entity can be enriched.

scope.zones table

Use the zones table to define areas of the network to be either included or excluded from the discovery process. A zone is typically defined as a list of varbinds. Varbinds are `name = value` pairs.

You can define multiple zones, and you can combine inclusion and exclusion zones. However, if you define a combination of inclusion and exclusion zones, the exclusion zones must be within the scope of the inclusion zones.

Table 94. scope.zones database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_Protocol`	• PRIMARY KEY • NOT NULL • Externally defined netProtocol data type	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`m_Action`	• NOT NULL • Externally defined filtAction data type	Integer	Action to perform for current zone: • 0: Undefined • 1: Include • 2: Exclude

Table 94. scope.zones database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_Zones`		List of type zone	A list of varbinds (name=value) that define the present discovery zone.
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

Example scope database configuration

The example OQL inserts into the scope database tables in this topic would be appended to the DiscoScope.cfg configuration file to configure the ncp_disco process when it starts.

Configuration of the scope.zones table

Use this information to understand how to configure the scope.zones table.

Creating two inclusion zones

This example configuration of the scope.zones table creates two inclusion zones for the current discovery. Both zones are defined using a single insert.

```
insert into scope.zones
(
        m_Protocol,
        m_Action,
        m_Zones
)
values
(
        1,
        1,
        [
            {
                m_Subnet="172.16.1.0",
                m_NetMask=24
            },
            {
                m_Subnet="172.16.2.*"
            }
        ]
);
```

The previous OQL insert specifies the following conditions:

- The network uses the Internet Protocol (m_Protocol=1).
- Any devices that fall into the present zone are to be included in the discovery (m_Action=1).
- The discovery includes:
  - Any device that falls within the 172.16.1.0 subnet (with a subnet mask of 24, that is, 24 bits turned on and 8 bits turned off, which implies a netmask of 255.255.255.0).
  - Any device with an IP address starting with 172.16.2, that is, in the 172.16.2.0 subnet with a mask of 255.255.255.0.

Creating a zone within a zone

Zones can be specified within zones: within a given inclusion zone, you can specify devices or subnets that are not to be detected. These devices are not pinged by the Ping finder or interrogated by the discovery agents. For example, you can define an include scope zone consisting of the Class B subnet 172.20.0.0/16, and completely contained within that zone you can specify an exclude scope zone consisting of the subnet 172.20.32.0/19. Finally, completely contained within the exclude scope zone you could specify an include scope zone 172.20.33.0/24.

```
// Include all IP addresses in this range
insert into scope.zones
(
        m_Protocol,
        m_Action,
        m_Zones
)
values
(
        1,
        1,
       [{m_Subnet = '172.20.0.0', m_NetMask = 16 }]
);

// Apart from the IP addresses in this range
insert into scope.zones
(
        m_Protocol,
        m_Action,
        m_Zones
)
values
(
        1,
        2,
        [{m_Subnet = '172.20.32.0' , m_NetMask = 19 }]
);
// Except for these IP addresses which we do want to include
insert into scope.zones
(
        m_Protocol,
        m_Action,
        m_Zones
)
values
(
        1,
        1,
        [{m_Subnet = '172.20.33.0' , m_NetMask = 24 }]
);
```

The previous OQL insert specifies three scope zones:

- All zones specify that the network uses the Internet Protocol (m_Protocol=1).
- Include and exclude zones are defined as follows:
  - Any devices that fall into the first zone, 172.20.0.0/16, are to be included in the discovery (m_Action=1).
  - Any devices that fall into the second zone, 172.20.32.0/19, which is completely contained within the first zone, are to be excluded from the discovery (m_Action=2).
  - Any devices that fall into the third zone, 172.20.33.0/24, which is completely contained within the second zone, are to be included in the discovery (m_Action=1).

Preventing the detection of devices with a filter

This example insert defines a detection filter. Since there must only be one insert into the scope.detectionFilter table, multiple conditions for IP must be defined using a single insert. The conditions of the filter can be combined using the Boolean OQL keywords AND and OR.

```
insert into scope.detectionFilter
(
        m_Protocol, m_Filter
```

```
)
values
(
        1,
        "(
                ( m_UniqueAddress <> '10.10.63.234' )
                AND
                ( m_ObjectId not like '1\.3\.6\.1\.4\.1\..*' )
        )"
);
```

The above example filter ensures that only the following are further interrogated by the discovery:

- Devices that do not have the IP address `10.10.63.234`.
- Devices that do not have the Object ID `1.3.6.1.4.1.*`.

In the above example, the backslash (\) is used in conjunction with the `not like` comparison to escape the `.` character, which would otherwise be treated as a wildcard.

Restricting instantiation based on entity name

This example insert defines an instantiation filter, also known as a postdiscovery filter. This example prevents the instantiation of devices that match a given entity name.

The filter (m_Filter) uses topology data in the ncimCache format.

Note: To ensure that alerts are not raised for *interfaces* that are excluded by the instantiation filter, you must set the `RaiseAlertsForUnknownInterfaces` variable. To this, perform the following steps:

1. Edit the `$NCHOME/etc/precision/NcPollerSchema.cfg` configuration file.
2. Add the following line to the file:

```
update config.properties set RaiseAlertsForUnknownInterfaces = 0;
```

Restricting instantiation of a chassis based on entity name

The following example postdiscovery filter restricts instantiation of a chassis and its contents.

```
insert into scope.instantiateFilter
(
    m_Filter
)
values
(
    "
    (
        BASENAME != 'jane'
    )
    "
);
```

Restricting instantiation of mutiple chassis

The following example postdiscovery filter restricts instantiation of a chassis and its contents.

```
insert into scope.instantiateFilter
(
    m_Filter
)
values
(
    "
    (
        snmpSystem->SYSDESCR NOT LIKE ' device'
    )
    "
);
```

Access databases

There are several databases that control access to network devices: snmpStack database and telnetStack database.

snmpStack database

The snmpStack database defines the operation of the SNMP helper.

Description

The snmpStack database in defined in the `SnmpStackSchema.cfg` file.

snmpStack.accessParameters database table

The snmpStack.accessParameters database table configures the way that the SNMP helper handles the retrieval of large non-scalar variables for particular devices or subnets.

Description

Any values inserted into this table override the values for `m_GetNextBoundary` and `m_GetNextSlowDown` that have been specified in the `snmpHelper.configuration` table.

Schema

The snmpStack.accessParameters database table schema is described in the following table:

Table 95. snmpStack.accessParameters database table schema

Column name	Constraints	Data type	Description
m_GeneralSlowDown	NOT NULL	Integer	The general amount by which to delay a request (in milliseconds). A general slow down must only be used where absolutely necessary as it can significantly increase the overall discovery time.
m_GetNextBoundary	NOT NULL	Integer	When retrieving a particular non-scalar SNMP variable from a device, this is the minimum number of GetNext requests to be issued before the delay specified by `m_GetNextSlowDown` is introduced.
m_GetNextSlowDown	NOT NULL	Integer	The delay (in milliseconds) to introduce between each SNMP GetNext request when the number of separate GetNext requests issued while retrieving a particular non-scalar SNMP variable exceeds `m_GetNextBoundary`.
m_NetAddress	NOT NULL	Text	The IP address on which to override the boundary and slowdown values.

Table 95. snmpStack.accessParameters database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_NetMask		Text	The netmask. If no netmask is specified, `m_NetAddress` is taken to be a single IP address. If a netmask is specified, `m_NetAddress` is taken to be a subnet address.
m_SnmpPort		Integer	The SNMP port on the target device, or target devices if the device access configuration specified by this record is applicable to a subnet. If no value is specified for m_SnmpPort, then the value defaults to the standard SNMP 161 port.

snmpStack.multibyteObjects table

The snmpStack.multibyteObjects table defines MIB objects that are checked to see if they are multibyte strings.

Description

Sending a raw ASCII string back to the helper server can cause problems if the string contains characters with special meaning in ASCII. If the MIB objects contain multibyte strings, the SNMP helper encodes them.

Schema

The snmpStack.multibyteObjects database table schema is described in the following table:

Table 96. snmpStack.multibyteObjects database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_ObjectName	NOT NULL	Text	The MIB object name to be checked.

Related reference

snmpStack.conversionCfg database table

The snmpStack.conversionCfg database table configures the SNMP Helper to replace characters that are not allowed in the locale of the NCIM database with the question mark character: '?'.

snmpStack.conversionCfg database table

The snmpStack.conversionCfg database table configures the SNMP Helper to replace characters that are not allowed in the locale of the NCIM database with the question mark character: '?'.

Description

The SNMP Helper substitutes characters on only those objects that are configured in the snmpStack.multibyteObjects table.

Inserts into this database table are configured in the `SnmpStackSchema.cfg` file.

Schema

The snmpStack.conversionCfg database table schema is described in the following table:

Table 97. snmpStack.conversionCfg database table schema

Column name	Constraints	Data type	Description
`m_SubstituteInvalidUTF8`	NOT NULL	Integer	If set to 1, the SNMP Helper replaces characters that are not allowed in the locale of the NCIM database with the question mark character: '?'. If set to 0, no action is taken on invalid characters.

Related reference

snmpStack.multibyteObjects table

The snmpStack.multibyteObjects table defines MIB objects that are checked to see if they are multibyte strings.

snmpStack.verSecurityTable database table

The snmpStack.verSecurityTable maps an IP or subnet address with an SNMP version (1, 2, or 3).

Description

The security parameters must be configured, as specified by the SNMP version, in order to gain SNMP access to network devices. An example of this is the use of community strings for SNMP versions 1 and 2, as well as the specification of the different security levels offered by SNMP V3.

Schema

The snmpStack.verSecurityTable database table schema is described in the following table:

Table 98. snmpStack.verSecurityTable database table schema

Column name	Constraints	Data type	Description
m_AccessLevel		Integer	The SNMP access level. Possible values are: • 1 - Read • 2 - Read/write The default value is 1.
m_EncryptedPwd		Integer	Whether the password stored in the configuration file is encrypted: • 1 - Encrypted • 2 - Unencrypted

Table 98. snmpStack.verSecurityTable database table schema (continued)

Column name	Constraints	Data type	Description
m_IpOrSubNetVer		Text	The IP or subnet address to which the device access configuration specified by this record is applicable. The interpretation of this field as an IP or a subnet address is dependent on the value specified in the `m_NetMaskBitsVer` field.
m_NetMaskBitsVer		Integer	The subnet mask for the address specified by the `m_IpOrSubNetVer` field. If this field is set to 32 then `m_IpOrSubNetVer` is taken as a single IP address.
m_NumRetries		Integer	The number of times to retry the request.
m_Password		Text	The password to try for this IP or subnet address; for example, community string.
m_Protocol		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
m_SecurityName		Text	The SNMP V3 security password.
m_SnmpPort		Integer	The SNMP port on the target device, or target devices if the device access configuration specified by this record is applicable to a subnet. If no value is specified for `m_SnmpPort`, then the value defaults to the standard SNMP 161 port.
m_SNMPVer3Details		Object of type V3SecInfo	An object representation of the authpassword and/or privpassword details for SNMP V3.
m_SNMPVer3Level		Integer	Integer representation of the SNMP V3 security level.

Table 98. snmpStack.verSecurityTable database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_SNMPVersion		Integer	The SNMP version that this configuration applies to. • 0: SNMP V1 • 1: SNMP V2 • 2: SNMP V3
m_TimeOut		Integer	The maximum time to wait for a reply from a device, in milliseconds.
m_Type		Integer	An integer classification of the password type; for example: (2) SNMP Get password.

telnetStack database

The telnetStack database defines the Telnet access parameters for devices.

Description

The telnetStack database is defined in the TelnetStackSchema.cfg file.

telnetStack.passwords database table

The telnetStack.passwords database table defines the Telnet access parameters for devices.

Schema

The telnetStack.passwords database table schema is described in the following table:

Table 99. telnetStack.passwords database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_AccessPort		Integer	The port on which to access the device.
m_ConPrompt		Text	Console prompt to expect from remote device. Default = "^.*[a-zA-Z0-9].*[$%>#]$".
m_EncryptedPwd		Integer	Whether the password stored in the configuration file is encrypted: • 1 - Encrypted • 2 - Unencrypted
m_IpOrSubNet		Text	IP or subnet address depending on value of m_NetMaskBits.
m_LogPrompt		Text	Login prompt to expect from remote device. Default = ".*ogin:.*".

Table 99. telnetStack.passwords database table schema (continued)

Column name	Constraints	Data type	Description
m_NetMaskBits		Integer	The subnet mask. If set to 32, m_IpOrSubNet is taken to be a single IP address.
m_Password	NOT NULL	Text	The password to try for this subnet or IP address. Default = "\n" (carriage return).
m_PreferSSHv1		Integer	Fix Pack 2 A boolean flag that indicates whether SSHv1 is used if a device supports SSH v1 and v2. Possible values are: • 0: Use SSHv2 support. • 1: Prefer SSHv1 support. If no value is specified for `m_PreferSSHv1`, then the value defaults to 0. The `m_PreferSSHv1` setting functions only when `m_SSHSupport` is enabled and the target device supports both SSH v1 and v2.
m_PrivAccessCmd		Text	The command for this device to enter into privileged mode.
m_PrivCommands		List type text	Which commands require privileged access.
m_PrivConPrompt		Text	The regular expression for the console prompt in privileged mode.
m_PrivPassword		Text	The password to enter privileged mode.
m_PrivPwdPrompt		Text	The regular expression for the password prompt in privileged mode.
m_Protocol		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
m_PwdPrompt		Text	Password prompt to expect from remote device. Default = ".*assword:.*".
m_SSHSupport		Boolean Integer	Flag to indicate whether or not to use SSH support for this device: • 1: Use SSH support for this device. • 0: Do not use SSH support for this device. If no value is specified for m_SSHSupport, then the value defaults to 0, that is, no SSH support.

Table 99. telnetStack.passwords database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_TimeOut		Integer	The time to wait for a response from the device, in milliseconds.
m_Username		Text	The username to try for this subnet or IP address. Default = "".

Process management databases

On startup, the discovery engine, ncp_disco, populates the agent and stitcher databases with information extracted from the discovery agent and discovery stitcher files. While operating, ncp_disco scans for alterations to the agent and stitcher files and updates the agent and stitcher databases if necessary. The frequency of scans is set in the disco database.

The agents and stitchers databases contain definition and configuration information for the agents and stitchers, such as a list of the types of devices that are sent to any given agent. The information in these databases is extracted by the discovery engine from the following directories:

- /precision/disco/agents
- /precision/disco/stitchers

The stitchers databases also contain information about when any given stitcher is triggered; for example, "start stitcher X upon completion of agent Y," or "start stitcher X upon the insertion of an entry into database table Z." It is therefore possible to start stitchers on demand by inserting their name into the appropriate actions table using OQL. The necessary agents are started automatically when a device is inserted into the despatch table of the agent.

Configuring the data flow: starting stitchers on-demand

The information extracted by DISCO contains the full definitions of the agents and stitchers, which includes the trigger conditions. By modifying the trigger conditions, you can modify the data flow of the discovery process.

You can start the discovery cycle from any point within the configured data flow by placing a stitcher into the actions table of the stitchers database.

agents database schema

The agents database is defined in `$NCHOME/etc/precision/DiscoSchema.cfg`. Its fully qualified database table names are: agents.definitions; agents.victims; agents.status

agents.definitions table

The agents.definitions table contains scheduling information for every discovery agent, extracted from the information in the discovery agent file.

Table 100. agents.definitions database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Name	• PRIMARY KEY • NOT NULL	Text	Name of the agent.

Table 100. agents.definitions database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_Type	Externally defined agentType data type	Integer	Agent Type: • 0: Undefined • 1: Precompiled • 2: Text defined • 3: Combination
m_Text	NOT NULL	Text	Textual description of agent rules.
m_ExecuteOn		Text	The host on which to execute the agent.
m_Phase	Default = 1	Integer	The discovery phase by the end of which the agent is expected to complete.
m_UpdTime		Long integer	The time of the last modification, which determines whether the agent has changed since its definition was stored.

agents.sourceInfo table

The agents.sourceInfo table holds information on the types of data source used by an agent.

Table 101. agents.sourceInfo database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_DiscoveryProtocol	NOT NULL	Text	The protocol used by the agent to get data from this source. Possible values are: • `Unknown` • `Other` • `Manual` • `FlatFile` • `SNMP` • `Telnet` • `XML-RPC` • `VSphere` • `OtherJavaAPI` • `TL1` • `CORBA`
m_Name	• PRIMARY KEY • NOT NULL	Text	Name of the agent.
m_RequiredField		List type text	Lists a field that must exist in the main node record in order for the entity to be considered of this source.

Table 101. agents.sourceInfo database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_RequiredValue			Lists a value for the field specified in m_RequiredField that must exist in the main node record in order for the entity to be considered of this source.
m_Source	NOT NULL	Text	The source of the data. Possible values are: • Unknown: source of the chassis is unknown. • Other: other source. • TopologyEditor: chassis was manually added using the Topology Editor. • PresetLayer: chassis was manually set using the PresetLayer stitcher. • Agent: chassis was discovered as part of the standard discovery process. • Collector: chassis was discovered as part of the EMS discovery process.

agents.status table

The agents.status table contains information about the present status of the agent.

Table 102. agents.status database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_CompletionPhase		Integer	The discovery phase in which the agent completes.
m_Name	• PRIMARY KEY • NOT NULL • UNIQUE	Text	Name of the agent.
m_NumConnects	Default = 0	Integer	The number of times that DISCO has connected to the agent.
m_State	Externally defined agentState data type. Default = 0	Integer	The current state of the agent: • 0: Undefined • 1: Not running • 2: Start up • 3: Running • 4: Finished • 5: Died

agents.victims table

The agents.victims table contains an extraction of the criteria that determine which devices get sent to the agent.

Table 103. agents.victims database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	Name of the agent.
`m_Filter`		Text	The filter condition that determines which devices are sent to the agent.

Stitchers database schema

The stitchers database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. Its fully qualified database table names are: stitchers.triggers; stitchers.status; stitchers.actions.

stitchers.triggers table

The stitchers.triggers table contains an extraction of the criteria that determine the trigger for the stitcher.

Table 104. stitchers.triggers database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	Name of the stitcher.
`m_Type`		Integer	The type of stitcher trigger: • 0: Undefined • 1: On the completion of some other activity; for example, another stitcher or a discovery phase • 2: On a table insert • 3: On demand • 4: On a timer
`m_Trigger`	Externally defined ruleTrigger data type	Object	Description of the stitcher trigger.

stitchers.status table

The stitchers.status table contains the information about the present status of the stitcher.

Table 105. stitchers.status database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	Name of the stitcher.
`m_State`	Externally defined stchrState data type Default = 0	Integer	The current state of the stitcher: • 0: Undefined • 1: Start up • 2: Running • 3: Finished • 4: Not maintained (the stitcher is not having its state maintained)

stitchers.actions table

If a stitcher is inserted into the stitchers.actions table, DISCO runs the stitcher. Once the stitcher has completed, its entry is deleted from the stitchers.actions table. Any stitchers triggered to execute from the stitcher that has been inserted, or upon completion of the stitcher, are also executed.

You can also configure other actions to take place on completion of the stitcher, so that the discovery cycle completes from that point onwards.

Table 106. stitchers.actions database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL	Text	Name of the stitcher.

Subprocess databases

The finders, Details, and agent databases are used during the discovery by the discovery engine subprocesses to store information retrieved from the network. The databases are defined within the configuration file, DiscoSchema.cfg.

The subprocess databases include:

- The finders database, which is used by the finders to store information about device existence.
- The Details database, which is used by the Details agent to store basic device information.
- The discovery agent databases, which are created using a template.

The finders, Details and AssocAddress agents must always be run, so their databases are defined in the DiscoSchema.cfg configuration file. The databases for the rest of the discovery agents are created based on a template that is defined in the DiscoSchema.cfg configuration file.

finders database schema

The finders database is defined in $NCHOME/etc/precision/DiscoSchema.cfg.

The fully qualified database table names of the finders database are:

- finders.despatch
- finders.returns
- finders.pending
- finders.processing
- finders.rediscovery

The finders database is the central monitoring and management point for finders operating during discovery. The finders discover the existence of devices and report these devices back to the finders database, but do not discover connections.

Network entities reported by the finders are usually sent to the Details agent for retrieval of basic device information, although the discovery data flow is fully configurable.

finders.despatch table

The finders.despatch table contains a record of all the requests sent to the finders and the current status of the requests.

Table 107. finders.despatch database table schema

Column name	Constraints	Data type	Description
`m_Finder`	• PRIMARY KEY • NOT NULL	Text	The name of the finder responsible for the request.
`m_FindRequest`	• PRIMARY KEY • UNIQUE • NOT NULL	Text	The OQL request sent to the finder named above.
`m_Request Status`		Integer	The current status of the request sent to the finder.

finders.returns table

When a finder finds a device, it returns the information to the finders.returns table, provided that the discovery is still in the device discovery phase, that is, data collection phase one. If the discovery is in the blackout state, the finders return the information to the pending table.

The returns table serves as a transfer point, notifying the system that a device exists. By default, a stitcher sends the device information to the Details agent to discover basic device information.

Table 108. finders.returns database table schema

Column name	Constraints	Data type	Description
`m_UniqueAddress`	• PRIMARY KEY • UNIQUE • NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
`m_Name`		Text	The unique name of the network entity.
`m_Creator`		Text	The finder that created this record.

Table 108. finders.returns database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_Protocol`		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device

finders.pending table

The pending table accepts device information when the returns table has been locked out by DISCO. The returns table has to be locked during data processing because even though the data collection stage has completed, it does not necessarily mean that all the devices on the network have been discovered.

Network entities that have been sent to the pending table are processed after the current discovery cycle has been completed.

Table 109. finders.pending database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_UniqueAddress`	• PRIMARY KEY • UNIQUE • NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
`m_Name`		Text	The unique name of the network entity.
`m_Creator`		Text	The finder that created this record in the table.
`m_Protocol`		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device

Table 109. finders.pending database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

finders.processing table

The processing table contains a record of all the discovered entities that are currently being processed by DISCO. Any device that has been reported to the returns table and is awaiting the next action to take place has an entry in the processing table.

Table 110. finders.processing database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_UniqueAddress`	• PRIMARY KEY • UNIQUE • NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
`m_Name`		Text	The unique name of the network entity.
`m_Creator`		Text	The finder that created this record in the table.
`m_Protocol`		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

finders.rediscovery table

The rediscovery table can hold nodes and subnets that you want to rediscover. Any device inserted into this table is sent to the Ping finder for processing.

Table 111. finders.rediscovery database table schema

Column name	Constraints	Data type	Description
`m_Address`	• PRIMARY KEY • NOT NULL	Text	The IP address of the discovered network entity.
`m_RequestType`		Int	The type of IP address: • 1: Individual • 2: Subnet
`m_NetMask`		Text	The net mask if the address refers to a subnet.
`m_Protocol`	NOT NULL	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6

CollectorDetails database schema

The `CollectorDetails` database is defined in `$NCHOME/etc/precision/DiscoSchema.cfg`. Its fully qualified database table names are: `CollectorDetails.despatch`; `CollectorDetails.returns`.

The CollectorDetails agent retrieves basic information about devices discovered by the collector finders when information from the collector finders is placed in the `despatch` table. The CollectorDetails agent retrieves the appropriate device information and places the results in the returns table.

CollectorDetails.despatch table

The despatch table contains basic information about devices that have been detected by the collector finders. When data is placed in this table, the CollectorDetails agent automatically interrogates the network for more detailed device information.

Table 112. CollectorDetails.despatch database table schema

Column name	Constraints	Data type	Description
`m_UniqueAddress`	• PRIMARY KEY • NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
`m_ManagerId`	NOT NULL	Text	Identifies the manager of the device. Takes the value "" if device is accessed directly.
`m_Name`		Text	Unique name of an entity on the network.

Table 112. CollectorDetails.despatch database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_Protocol`		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
`m_DespatchTime`		Timestamp	Date and time the network was interrogated by the CollectorDetails agent.
`m_ExtraInfo`	Externally defined vblist data type	Object	Any extra information.

CollectorDetails.returns table

The returns table holds detailed device information retrieved by the CollectorDetails agent. Information inserted into this table is automatically processed by the stitchers so that the device connectivity can be discovered by the appropriate discovery agent.

Table 113. CollectorDetails.returns database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_Name`		Text	Unique name of an entity on the network.
`m_UniqueAddress`	NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
`m_ManagerId`	NOT NULL	Text	Identifies the manager of the device. Takes the value "" if device is accessed directly.

Table 113. CollectorDetails.returns database table schema (continued)

Column name	Constraints	Data type	Description
m_Protocol		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
m_ObjectId		Text	Textual representation of the device class (an ASN.1 address).
m_Description		Text	Value of sysDescr MIB variable of the entity.
m_HaveAccess	Externally defined Boolean data type	Integer	Flag indicating whether there is SNMP access to the device: • 1: Have access • 0: No access
m_UpdAgent		Text	The agent that updated this device.
m_LastRecord	Externally defined Boolean data type	Boolean integer	A flag indicating whether this is the last record for this entity (that is, whether the entity has been completely processed): • 1: True • 0: False
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
m_ReturnTime		Timestamp	Date and time this value was returned.
m_ExtraInfo	Externally defined vblist data type	Object	Any extra information.

Details database schema

The Details database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. Its fully qualified database table names are: `Details.despatch`; `Details.returns`.

The Details agent retrieves basic information about devices discovered by the finders when information from the finders is placed in the despatch table. The Details agent retrieves the appropriate device information and places the results in the returns table.

A stitcher takes the information from the Details.returns table and sends it to the Associated Address agent and ultimately the appropriate discovery agent.

details.despatch table

The despatch table contains basic information about devices that have been detected by the finders. When data is placed in this table, the Details agent automatically interrogates the network for more detailed device information.

Table 114. Details.despatch database table schema

Column name	Constraints	Data type	Description
`m_UniqueAddress`	• PRIMARY KEY • NOT NULL	Text	The IP address of the discovered network entity.
`m_Name`		Text	Unique name of an entity on the network.
`m_ManagerId`	NOT NULL	Text	Identifies the manager of the device. Takes the value "" if device is accessed directly.
`m_Protocol`		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

details.returns table

The returns table holds detailed device information retrieved by the Details agent. Information inserted into this table is automatically processed by the stitchers so that the device connectivity can be discovered by the appropriate discovery agent.

Table 115. Details.returns database table schema

Column name	Constraints	Data type	Description
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
m_Description		Text	Value of sysDescr MIB variable of the entity.
m_ExtraInfo	Externally defined vblist data type	Object	Any extra information.
m_HaveAccess	Externally defined Boolean data type	Integer	Flag indicating whether there is SNMP access to the device: • 1: Have access • 0: No access
m_LastRecord	Externally defined Boolean data type	Boolean integer	A flag indicating whether this is the last record for this entity (that is, whether the entity has been completely processed): • 1: True • 0: False
m_ManagerId	NOT NULL	Text	Identifies the manager of the device. Takes the value "" if device is accessed directly.
m_Name		Text	Unique name of an entity on the network.
m_ObjectId		Text	Textual representation of the device class (an ASN.1 address).
m_Protocol		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6

Table 115. Details.returns database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_UniqueAddress`	NOT NULL	Text	The IP address of the discovered network entity.
`m_UpdAgent`		Text	The agent that updated this device.

Finders databases

Finders determine device existence. Each of the finders uses a different method to discover network devices. You can enable finders for your discovery by configuring them as managed processes of DISCO in their respective configuration files. Finders are automatically launched at the appropriate time, provided that CTRL is running.

Each finder must be configured by editing its configuration file. The finders discover the existence of devices and report these devices back to the finders database, but do not discover connections.

Note that the finders database is distinct from the databases that are associated with the individual finders.

The finders are described in the table below, with their executable name and the location of their configuration file. $NCHOME is the environment variable that contains the path to the netcool directory.

Table 116. Description of the finders			
Finder	**Executable**	**Configuration file**	**Description**
Ping	ncp_df_ping	$NCHOME/etc/precision/DiscoPingFinderSchema.cfg $NCHOME/etc/precision/DiscoPingFinderSeeds.cfg	Makes a simple ICMP echo request for broadcast or multicast addresses, individual IP addresses, or all devices on a subnet.
File	ncp_df_file	$NCHOME/etc/precision/DiscoFileFinderSchema.cfg $NCHOME/etc/precision/DiscoFileFinderParseRules.cfg	Parses a file, such as `/etc/hosts`, to retrieve a list of devices to find devices on the network.
Database	ncp_df_dbentry	$NCHOME/etc/precision/DiscoDBEntryFinderSchema.cfg	Reads a database in order to retrieve a list of devices to find on the network.
Collector	ncp_df_collector	$NCHOME/etc/precision/DiscoCollectorFinderSchema.cfg $NCHOME/etc/precision/DiscoCollectorFinderSeeds.cfg	An EMS collector is a software module that retrieves and stores topology data from an Element Management System (EMS). The Collector finder queries a collector and gets a list of IP addresses managed by the EMS associated with that collector.

collectorFinder database

The collectorFinder database defines the operation of the Collector finders.

Description

The collectorFinder database is defined in the `DiscoCollectorFinderSchema.cfg` configuration file. It has the following tables:

- collectorFinder.collectorRules
- collectorFinder.configuration

collectorFinder.collectorRules database table

The collectorFinder.collectorRules database table configures the operation of the Collector finder.

Description

You can override some of the settings for particular collectors in the collectorFinder.configuration table. The collectorRules table can contain multiple records.

Schema

The collectorFinder.collectorRules database table schema is described in the following table:

Table 117. collectorFinder.collectorRules database table schema

Column name	Constraints	Data type	Description
`m_Host`		Text	The host address on which the collector is running. This field is NOT NULL only if the collector is running on a different host to Network Manager. This field may be configured for both a discovery and a rediscovery.
`m_Port`	• PRIMARY KEY • NOT NULL	Text	The port on which the collector is listening. If the collector is running on the same host as Network Manager, then this is a Network Manager port. This field may be configured for both a discovery and a rediscovery.

Table 117. collectorFinder.collectorRules database table schema (continued)

Column name	Constraints	Data type	Description
`m_RequestType`		Integer	Flag denoting which topology data to download from the data source. This flag works together with the m_Address and m_NetMask fields. The flag takes the following values: • 0: Rediscover all devices. All devices retrieved by the collector are discovered. The m_Address and m_NetMask fields are ignored. • 1: Rediscover single device. Only one of the devices retrieved by the collector is discovered. The m_Address field specifies the device and the m_NetMask fields is ignored. • 2: Rediscover subnet. One of the subnets retrieved by the collector is discovered. The m_Address field specifies the subnet and the m_NetMask field specifies the subnet mask. This field is configured for a rediscovery only.
`m_DataSourceId`		Integer	Limits rediscovery to a single data source supported by the collector. This field is rarely used as a collector usually only supports a single data source. This field is configured for a rediscovery only.
`m_Address`		Text	Used in conjunction with the m_RequestType and m_NetMask fields when specifying a device or subnet to rediscover. See the entry for m_RequestType for more information. This field is configured for a rediscovery only.
`m_NetMask`		Text	Used in conjunction with the m_RequestType and m_Address fields when specifying a device or subnet to rediscover. See the entry for m_RequestType for more information. This field is configured for a rediscovery only.
`m_NumRetries`		Integer	Number of retries to issue an RPC XML request to the collector. Setting this field is optional. If set, this field overrides the default specified in the collectorFinder.configuration table. This field may be configured for both a discovery and a rediscovery.

collectorFinder.configuration database table

The collectorFinder.configuration table specifies the general rules of the Element Management System (EMS) collector methodology and must only contain one record.

Schema

The collectorFinder.configuration database table schema is described in the following table:

Table 118. collectorFinder.configuration database table schema

Column name	Constraints	Data type	Description
m_NumThreads		Integer	The number of threads to be used by the Collector finder.
m_TimeOut		Integer	The maximum time to wait for a reply from a collector (the timeout).
m_NumRetries		Integer	The number of times to issue an RPC XML request to a collector.
m_MaxResponseSize		Integer	The maximum size for an XML-RPC response in bytes. **Note:** The default maximum response size might be too small when running a Collector-based discovery against Collectors that result in very large responses. In such cases, increase the maximum response size. To increase the maximum response size, set the **m_MaxResponseSize** parameter to a higher value. Make sure you set the same value for **m_MaxResponseSize** in both of the following files: • NCHOME/etc/precision/DiscoCollectorFinderSchema.cfg • NCHOME/etc/precision/DiscoXmlRpcHelperSchema.cfg

dbEntryFinder database

The dbEntryFinder database defines the operation of the Database finder.

Description

The dbEntryFinder database is defined in the DiscoDBEntryFinderSchema.cfg file. It has the following tables:

- dbEntryFinder.configuration
- dbEntryFinder.dbQueries

dbEntryFinder.configuration database table

You can configure the Database finder with the dbEntryFinder.configuration table, which specifies the number of threads to be used by the finder.

Schema

The dbEntryFinder.configuration database table is described in the following table.

Table 119. dbEntryFinder.configuration database table schema

Column name	Constraints	Data type	Description
`m_NumThreads`	NOT NULL	Integer	The number of threads to be used by the Database finder.

dbEntryFinder.dbQueries database table

By configuring inserts into the dbEntryFinder.dbQueries table, you can specify the SQL queries to run to retrieve device details from the database that stores discovery seed details. You can also specify optional mapping parameters that define how to map the device details retrieved from the database to the finders.returns or finders.discovery tables.

Schema

The dbEntryFinder.dbQueries database table schema is described in the following table:

Table 120. dbEntryFinder.dbQueries database table schema

Column name	Constraints	Data type	Description
`m_DBEntryName`	• NOT NULL • UNIQUE	Text	The unique name of this database entry.
`m_DbId`		Text	Identifier of the database that contains the device details, as defined in the `DbLogins.`*`DOMAIN`*`.cfg` configuration file.
`m_TriggerType`	NOT NULL	Integer	Indicates which trigger type invokes this query: • 1 Full discovery • 2 Partial rediscovery • 3 Forced rediscovery
`m_Query`		Text	Code of the query.
`m_Parameters`		List of atoms	Optional parameters to pass to the query.

Table 120. dbEntryFinder.dbQueries database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_Mapping		List of atoms	Optional mapping parameters that define how to map the fields returned from the query into fields expected by the Discovery engine, ncp_disco. These parameters map data into target database tables as follows: • If m_TriggerType is 1 or 2, then map the data to the finders.returns table. • If m_TriggerType is 3, then map the data to the finders.rediscovery table.

fileFinder database

The fileFinder database defines the operation of the File finder.

Description

The fileFinder database is defined in the DiscoFileFinderParseRules.cfg file. It has the following tables:

- fileFinder.configuration
- fileFinder.parseRules

fileFinder.configuration database table

You can configure the File finder with the fileFinder.configuration table, which specifies the number of threads to be used by the finder.

Schema

The fileFinder.configuration database table is described in the following table.

Table 121. fileFinder.configuration database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_NumThreads	NOT NULL	Integer	The number of threads to be used by the File finder.

fileFinder.parseRules database table

By configuring inserts into the fileFinder.parseRules table, you can specify the files to be parsed for a list of IP addresses of devices on the network.

Description

The fileFinder.parseRules table specifies the rules for file parsing.

A typical file that you would parse, for example, is the /etc/hosts file on the machine running DISCO. You can also seed the discovery by parsing the /etc/defaultrouter file.

Schema

The fileFinder.parseRules database table schema is described in the following table:

Table 122. fileFinder.parseRules database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_FileName`	• NOT NULL • UNIQUE	Text	The unique full path and filename of the file to be parsed, for example, `/etc/hosts`.
`m_Delimiter`		Text	The delimiter that separates the data fields in the file. Regular pattern matching expressions are also accepted as valid delimiters. **Note:** \t is not supported as a valid value for the <tab> character.
`m_ColDefs`		List of atoms	A list of rules that specify which variables to extract and the columns from which to get them.

pingFinder database

The pingFinder database defines the operation of the Ping finder.

Description

The pingFinder database is defined in the DiscoPingFinderSeeds.cfg file. It has the following tables:

- pingFinder.configuration
- pingFinder.pingFilter
- pingFinder.pingRules
- pingFinder.scope

pingFinder.configuration database table

The pingFinder.configuration table specifies the general rules of the ping methodology. The table must contain only one record.

Description

The pingFinder.configuration table allows you to configure the way devices are pinged, including enabling broadcast or multicast pinging. Although pinging of broadcast/multicast addresses allows devices to be discovered more quickly than other detection methods, it is sometimes less desirable to do so under certain network conditions, such as when the network is heavily congested. In general, you would ping broadcast addresses on an unknown sparsely populated network. You must only ping multicast addresses where they have been set up on the network.

Schema

The pingFinder.configuration database table schema is described in the following table:

Table 123. pingFinder.configuration database table schema		
Column name	**Data type**	**Description**
`m_NumThreads`	Integer	The number of threads to be used by the Ping finder.

Table 123. pingFinder.configuration database table schema (continued)		
Column name	**Data type**	**Description**
`m_TimeOut`	Integer	The maximum time to wait for a reply from a pinged address (the timeout).
`m_InterPingTime`	Integer	The interval between pinging the addresses in a subnet.
`m_NumRetries`	Integer	The number of times a device is to be re-pinged.
`m_Broadcast`	Integer	Flag used to enable or disable broadcast address pinging: • 1: Enable • 0: Disable
`m_Multicast`	Integer	Flag used to enable or disable multicast address pinging: • 1: Enable • 0: Disable

pingFinder.pingFilter database table

The pingFinder.pingFilter table can be used to exclude particular devices or subnets from being pinged by the Ping finder.

Description

You may wish to exclude certain interfaces, such as ISDN and modem interfaces, because pinging these interfaces generates phone calls, which costs money. If you configure the Ping finder to use both the scope.zones table and the pingFinder.pingFilter table, the Ping finder pings those devices or subnets it has been seeded with if they are within either the discovery scope or the Ping finder scope.

Schema

The pingFinder.pingFilter database table schema is described in the following table:

Table 124. pingFinder.pingFilter database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_Protocol`	• PRIMARY KEY • NOT NULL • Externally defined netProtocol data type	Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6

Table 124. pingFinder.pingFilter database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_Action	• NOT NULL • Externally defined netProtocol data type	Integer	Action to perform for current zone: • 0: Undefined • 1: Include • 2: Exclude
m_Zones		List of type zone	A list of varbinds (name=value) that define the present zone.
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

pingFinder.pingRules database table

The pingFinder.pingRules table specifies the different addresses and subnets to be pinged by the Ping finder.

Description

The pingRules table can contain multiple records.

Schema

The pingFinder.pingRules table is described in the following table.

Table 125. pingFinder.pingRules database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Address	• PRIMARY KEY • NOT NULL	Text	The address to ping.
m_RequestType		Integer	Flag denoting address type: • 1: Individual • 2: Subnet
m_NetMask		Text	The subnet mask. If a value is specified for this field, it automatically implies that the address is a subnet address.
m_TimeOut		Integer	Maximum time to wait for response. This value overrides the default timeout specified in the configuration table.
m_NumRetries		Integer	Maximum number of times to reattempt the ping. This value overrides the default value.

pingFinder.scope database table

The pingFinder.scope table defines the scope of the Ping finder.

Description

You can use the pingFinder.scope table to configure the way the Ping finder checks whether it is allowed to ping a particular device. You can exclude particular devices or subnets from being pinged by the Ping finder.

Schema

The pingFinder.scope database table schema is described in the following table:

Table 126. pingFinder.scope database table schema

Column name	Constraints	Data type	Description
`m_UseScope`		Integer	Flag denoting whether or not to use the entries in the `scope.zones` table when deciding which devices to ping: • 0: The Ping finder ignores the scope.zones table when deciding which devices to ping. • 1: This is the default value. The Ping finder uses the scope.zones table to check which devices can be pinged. If you are performing an unscoped discovery, that is, a discovery without any entries in the scope.zones table, then it is preferable to set m_UseScope to zero to reduce processing load.
`m_UsePingEntries`		Integer	Flag denoting whether or not to use the entries in the pingFinder.pingFilter table when deciding which devices to ping: • 0: This is the default value. The Ping finder ignores any entries in the pingFinder.pingFilter table when deciding which devices can be pinged. • 1: The Ping finder checks the pingFinder.pingFilter table before it pings a particular device to see if the device can be pinged.

The Helper Server databases

When the Helper Server starts, it creates a database for each helper that is to be run.

Tip: It is good practice to configure the Helper Server to start automatically by making the appropriate OQL insertion into the services.inTray table of CTRL. Alternatively, you can start the Helper Server manually with the **ncp_d_helpserv** command on the command line.

ARPhelper database

The ARPHelper database configures the operation of the ARP helper, stores information about the requests the ARP helper makes from the network.

The ARPHelper database is defined in `NCHOME/etc/precision/ DiscoHelperServerSchema.cfg`.

ARPhelper.ARPHelperConfig table

The ARPhelper.ARPHelperConfig database table configures the general operation of the ARP helper.

The ARPhelper.ARPHelperConfig database table is described in the following table.

Table 127. ARPhelper.ARPHelperConfig database table schema

Column name	Constraints	Data type	Description
`m_HelperDbTimeout`	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires in the absence of any activity.
`m_HelperReqTimeout`		Long64	The helper request timeout, that is, how long before each request expires.
`m_HelperStartupTimeout`		Long64	The default helper startup timeout, that is, the maximum time to wait for a helper to start up when requested.
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: (0) Do not use cache (1) Use cache
`m_HelperDoQueryVBs` Optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.
`m_HelperDoNotQueryVBs` Optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides the value specified in `m_HelperDoWeQuery`.
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: (0) Do not store replies in database (1) Store replies in database
`m_HelperDoStoreVBs` Optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides the value of `m_HelperDoWeStore`.

Table 127. ARPhelper.ARPHelperConfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDoNotStoreVBs` Optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides the value of `m_HelperDoWeStore`.
`m_HelperDebugLevel` Optional		Integer	Sets the debug level of the helper, printing to `m_HelperLogfile`.
`m_HelperLogfile` Optional		Text	The full path and file for the logfile of the current helper.

The m_HelperDoWeQuery and m_HelperDoWeStore fields each have two related optional fields. A record entered into either m_HelperDoWeQuery or m_HelperDoWeStore is the default setting to which the helper responds if no records are entered into the optional fields. However, a record entered into either of the related optional fields overrides the default setting.

For example, if m_HelperDoWeQuery is set to query the network rather than the cache (that is, m_HelperDoWeQuery=0) and if an IP address of 192.168.0.1 is specified in m_HelperDoQueryVBs, then a request record where `m_IpAddress = 192.168.0.1` results in the cache being queried rather than the network. The network is only queried if the information is not currently held in the cache.

ARPhelper database configuration

The following example insert gives a typical ARP helper configuration.

```
insert into ARPHelper.ARPHelperConfig
(
        m_HelperDbTimeout,
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200, 1200, 90, 0, 0
);
```

ARPhelper.ARPHelperTable table

The ARPHelper.ARPHelperTable database table stores information about the requests the ARP helper makes from the network.

The ARPHelper.ARPHelperTable database table is described in the following table.

Table 128. ARPHelper.ARPHelperTable database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_AddressSpace`		Text	The address space of the device.
`m_HostIp`	NOT NULL	Text	IP address of the device to interrogate.

Table 128. ARPHelper.ARPHelperTable database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_HostMac		Text	The physical address of the device (MAC address).
m_HostMask		Text	The subnet mask of the host device to be interrogated.
m_HostSubnet		Text	Subnet of the host device to be interrogated.
m_Protocol		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
RivHelperDbTimeToDie		Long64	Indicates how long the requested information is to live within the Helper Server.
RivHelperRequestGetKey	NOT NULL	Text	A key interface to the databases of the Helper Server for Get requests.
RivHelperRequestReplyKey	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A unique key interface to the databases of the Helper Server for Reply requests.

ARPHelper.configuration table

The ARPHelper.configuration database table defines the number of threads the helper uses.

The ARPHelper.configuration database is described in the following table.

Table 129. ARPHelper.configuration database table schema			
Column Name	**Constraints**	**Data type**	**Description**
m_NumThreads	None	Integer	The number of threads to be used by the helper.

DNSHelper database

The DNSHelper database is defined in NCHOME/etc/precision/DiscoHelperServerSchema.cfg.

The DNSHelper database table stores information about the requests that the DNS helper makes from the network, and configuration information for the DNS helper.

DNSHelper.DNSHelperTable table

The DNSHelper.DNSHelperTable database table stores information about the requests that the ARP helper makes from the network.

The DNSHelper.DNSHelperTable database table is described in the following table.

Table 130. DNSHelper.DNSHelperTable database table schema

Column name	Constraints	Data type	Description
`m_HostName`		Text	The host name for this IP address.
`m_HostIp`		Text	The IP addresses for this host.
`RivHelperDbTimeToDie`		Long64	How long the requested information is to live within the Helper Server.
`RivHelperRequestGetKey`	NOT NULL	Text	A key for Get requests.
`RivHelperRequestOutput`		Atom	The response data.
`RivHelperRequestReplyKey`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A unique key for Reply requests.

DNSHelper.DNSHelperConfig table

The DNSHelper.DNSHelperConfig table holds configuration information for the DNS helper.

The DNSHelper.DNSHelperConfig table is described in the following table.

Table 131. DNSHelper.DNSHelperConfig database table schema

Column name	Constraints	Data type	Description
`m_HelperDbTimeout`	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires.
`m_HelperDebugLevel` optional		Integer	Sets the debug level of the helper, printing to m_Logfile.
`m_HelperDoQueryVBs` optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.

Table 131. DNSHelper.DNSHelperConfig database table schema (continued)

Column name	Constraints	Data type	Description
`m_HelperDoNotQueryVBs` optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides the value of m_HelperDoWeQuery.
`m_HelperDoNotStoreVBs` optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides m_HelperDoWeStore.
`m_HelperDoStoreVBs` optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides m_HelperDoWeStore.
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: • 0: Do not use cache • 1: Use cache
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: • 0: Do not store replies in database • 1: Store replies in database
`m_HelperLogfile` optional		Text	The full path and file for the logfile of the current helper.
`m_HelperReqTimeout`		Long64	The helper request timeout, that is, how long before each request expires.
`m_HelperStartupTimeout`		Long64	The default helper start-up timeout, that is, the maximum time to wait for a helper to start up when requested.

DNS helper database configuration

The following example insert shows a typical DNS helper configuration.

```
insert into DNSHelper.DNSHelperConfig
(
        m_HelperDbTimeout,
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200, 1200, 90, 0, 0
);
```

DNSHelper.configuration table

The DNSHelper.configuration database table configures the operation of the DNS Helper. This table must contain only one record

The DNSHelper.configuration table is described in the following table.

Table 132. DNSHelper.configuration database table schema

Column name	Constraints	Data type	Description
`m_NumThreads`		Integer	The number of threads to be used by the helper.
`m_MethodList`		List of text	An ordered list of the methods for name retrieval.

DNShelper.methods table

The DNShelper.methods database table holds information used by the DNS Helper to access devices.

The DNShelper.methods database table is described in the following table.

Table 133. DNShelper.methods database table schema

Column name	Constraints	Data type	Description
`m_FileName`		Text	The filename, if appropriate.
`m_FileOrder`		Integer	The order of the files: • 0: Name first, then IP address • 1: IP address, then name
`m_MethodName`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The name of the method.
`m_MethodType`		Integer	The type of the method: • 0: System • 1: DNS • 2: File

Table 133. DNShelper.methods database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_NameDomain		Text	Domain name; for example, abcd.com.
m_NameDomainList		Text	Contains a list of expected domain suffixes. If you expect the discovery to return some or all devices names with domain suffixes already appended, then you can specify a list of expected domain suffixes in this column. **Note:** The domain suffix value specified in m_NameDomain is not appended to any device names returned by the discovery that have any of the suffixes listed in m_NameDomainList.
m_NameServerAddr		Text	The IP address of the DNS server (specified as a text string). If no value is specified, /etc/resolv.conf is read.
m_TimeOut		Integer	Time out for the request in seconds.

PingHelper database

The PingHelper database is defined in NCHOME/etc/precision/DiscoHelperServerSchema.cfg.

PingHelper.configuration table

The PingHelper.configuration database table configures broadcast and multicast pinging.

Although pinging broadcast and multicast addresses allows devices to be discovered quicker than other detection methods, it is not advisable to do so under certain network conditions; for instance, when the network is heavily congested.

The PingHelper.configuration database table must contain only one record.

The schema of the PingHelper.configuration database table is described in the following table.

Table 134. PingHelper.configuration database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Broadcast		Integer	Flag used to enable or disable broadcast address pinging: • (1) Enable • (0) Disable
m_ICMPSrcPort			The ICMP source port.
m_InterPingTime		Integer	The time interval in milliseconds between successive ping attempts of subnet addresses.
m_NumRetries		Integer	The number of times a device is to be re-pinged.

Table 134. PingHelper.configuration database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_NumThreads		Integer	The number of threads to be used by the helper.
m_Multicast		Integer	Flag used to enable or disable multicast address pinging: • (1) Enable • (0) Disable
m_TimeOut		Integer	The maximum time to wait for a reply from a pinged address, in milliseconds. If you are running the TraceRoute agent you may need to increase this value, depending on network conditions.
m_UdpSrcPort		Integer	The UDP port to be used.

PingHelper.PingHelperConfig table

The PingHelper.PingHelperConfig database table configures the operation of the Ping helper.

The schema of the PingHelper.PingHelperConfig database table is described in the following table.

Table 135. PingHelper.PingHelperConfig database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDbTimeout`	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires.
`m_HelperDebugLevel` optional		Integer	Sets the debug level of the helper, printing to the file specified in m_HelperLogfile.
`m_HelperDoNotQueryVBs` optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides m_HelperDoWeQuery.
`m_HelperDoNotStoreVBs` optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides m_HelperDoWeStore.
`m_HelperDoQueryVBs` optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.

Table 135. PingHelper.PingHelperConfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDoStoreVBs` optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides m_HelperDoWeStore.
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: • 0: Do not use cache • 1: Use cache
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: • 0: Do not store replies in database • 1: Store replies in database
`m_HelperLogFile` optional		Text	The full path and file for the logfile of the current helper.
`m_HelperReqTimeout`		Long64	The helper request timeout that is, how long before each request expires.
`m_HelperStartupTimeout`		Long64	The default helper startup timeout, that is, the maximum time to wait for a helper to start up when requested to.

PingHelper.PingHelperConfig database table configuration

The following insert provides a typical example configuration of the PingHelper.PingHelperConfig database table.

```
insert into PingHelper.PingHelperConfig
(
        m_HelperDbTimeout,
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200, 1200, 90, 0, 0
);
```

PingHelper.PingHelperTable table

PingHelper.PingHelperTable database table configures the operation of the Ping helper.

The schema of the PingHelper.PingHelperTable database table is described in the following table.

Table 136. PingHelper.PingHelperTable database table schema

Column name	Constraints	Data type	Description
`m_HostIp`		Atom	IP address to ping.
`m_HostMask`		Text	The subnet mask of the address to ping.
`m_HostSubnet`		Text	Subnet of the IP address to ping.
`m_PingRequestType`		Integer	The type of ping request: • 1: Individual address • 2: Subnet
`m_PingResponseType`		Integer	Type of reply to the ping.
`m_PingRetries`		Integer	Number of retries for the ping.
`m_PingTimeout`		Integer	Maximum time to wait for reply.
`m_Protocol`		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`RivHelperDbTimeToDie`		Long64	How long the requested information is to live within the Helper Server.
`RivHelperRequestGetKey`	NOT NULL	Text	A key interface to the databases of the Helper Server for Get requests.

Table 136. PingHelper.PingHelperTable database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
RivHelperRequestReplyKey	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A key interface to the databases of the Helper Server for Reply requests.
RivHelperRequestOutput		Atom	The response data.

snmpHelper database

The snmpHelper database configures the operation of the SNMP Helper. This database is defined in NCHOME/etc/precision/DiscoSnmpHelperSchema.cfg.

snmpHelper.configuration table

The snmpHelper.configuration database table configures the operation of the SNMP Helper.

The snmpHelper.configuration database table is described in the following table.

Table 137. snmpHelper.configuration database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_ExponentialRetries	None	Integer	Number of times to try to reestablish SNMP communication with a device that has temporarily stopped responding
m_LogRequests	None	Boolean	Specifies whether the system should log details about each request serviced by the SNMP helper. If set to positive, this causes the SnmpHelperDebug.*DOMAIN*.Trace file to be created in the log directory.
m_NumIOThreads	None	Integer	The number of input/output threads that the engine should use. The default value is 10, and the maximum allowed value is 1000 but it is best practice to set this value no higher than 100.
m_NumRetries	None	Integer	The number of attempts to retrieve SNMP variable(s) from a device.
m_NumThreads	None	Integer	The number of threads to be used by the helper.
m_ReadStackConfigOnConnect	None	Boolean	Indicates whether to read (or reread) the contents of the SnmpStackSecurityInfo.cfg file when connecting to the Helper server, ncp_d_helpserv.
m_ResolveHostNames	None	Boolean	Specifies whether the SNMP helper **ncp_dh_snmp** should attempt to resolve a hostname.
m_TimeOut	None	Integer	The maximum time to wait for a reply from a device, in milliseconds.

snmpHelper.dependentInstanceFilter database table

The snmpHelper.dependentInstanceFilter database table is used to define interface filters that depend on other filters.

Schema

The snmpHelper.dependentInstanceFilter database table schema is described in the following table:

Table 138. snmpHelper.dependentInstanceFilter database table schema

Column name	Constraints	Data type	Description
m_InstanceFilter	not null primary key	text	The dependent filter. The syntax of the filter is `MIB_variable_name in (eval(list_type,'&MIB_table.MIB_entry'))` where *MIB_variable_name* must exist in *MIB_table*, and a filter on *MIB_table* has been defined in the snmpHelper.instanceFilter table.
m_ApplyToFilteredTable		text	This value is automatically derived from m_InstanceFilter. You must not configure inserts into this field.

snmpHelper.instanceFilter database table

The snmpHelper.instanceFilter database table configures SNMP interface filters for the SNMP helper.

Schema

The snmpHelper.instanceFilter database table schema is described in the following table:

Table 139. snmpHelper.instanceFilter database table schema

Column name	Constraints	Data type	Description
m_Filter Name	not null, primary key	text	The name of the interface filter. The name must be unique.

Table 139. snmpHelper.instanceFilter database table schema (continued)

Column name	Constraints	Data type	Description

m_DeviceFilter	not null	text	The device filter is applied to each discovered device to determine whether or not to apply the interface filter.
			The filter must be in the following form:
			`mibVariableName expression values`
			`[ optional_Boolean_operator expression optional_Boolean_operator .. ]`
			For example, the following are all valid filters:
			`// Apply the interface filter to only a specific type of device`
			`sysObjectID = '1.3.6.1.4.1.4874.1.1.1.1.3'`
			`// More complex example of the above`
			`sysObjectID = '1.3.6.1.4.1.4874.1.1.1.1.3' OR sysDescr LIKE 'ERX-1440'`
			`// Apply the interface filter only to devices in certain locations`
			`sysLocation in ( 'location1', 'location2' )`
			`//Apply the interface filter to all types of device.`
			`sysObjectID != ''`

m_DeviceFilterOids		list type text	Specifies any OIDs that need to be retrieved in order to allow the evaluation of the device filter defined in the m_DeviceFilter field. The OIDs are usually determined programatically from the value of m_DeviceFilter. You do not normally need to define the OIDs manually.

m_InstanceFilter		text	The interface filter to be applied to the filtered tables. The interface filter is only applied to devices that match the device filter. Only rows from those tables with interfaces that match this filter are returned.
			The filter must be in the following form:
			`mibVariableName expression values`
			`[ optional_Boolean_operator expression optional_Boolean_operator .. ]`
			For example, the following are all valid filters:
			`// Only interfaces with names like this are returned`
			`ifName like 'Gi0'`
			`// This filter is against 2 distinct tables (ifTable and ifXTable)`
			`with the requirement that these share a common index (ifIndex)`
			`ifName like 'Gi0' or ifDescr like 'FastEthernet'`
			`// Filter out interfaces of these types`
			`ifType not in ( 1, 53, 166 )`
			Restriction: You can configure inserts into only one of the m_InstanceFilter or m_InstanceFilterTable fields.
			Restriction:
			The MIB variable used in the interface filter must be from a table that is keyed on ifIndex, for example, from ifTable or ifXTable.

Table 139. snmpHelper.instanceFilter database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_InstanceFilterTable		text	Defines a table that is not to be queried. **Restriction:** You can configure inserts into only one of the m_InstanceFilter or m_InstanceFilterTable fields. **Restriction:** If you define a table that is not to be queried using the m_InstanceFilterTable field, you must not apply other filters to the same MIB table for the same device.
m_InstanceFilters		list type object type vblist	The m_InstanceFilters field contains the full collection of interface filters. The full interface filters are automatically generated from the contents of the m_InstanceFilter or m_InstanceFilterTable fields. **Restriction:** User-configured inserts into this field are not supported.

snmpHelper.SnmpHelperConfig table

The snmpHelper.SnmpHelperConfig database table configures the operation of the SNMP Helper.

The schema of the snmpHelper.SnmpHelperConfig database table is described in the following table.

Table 140. snmpHelper.SnmpHelperConfig database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDbTimeout`	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires.
`m_HelperDebugLevel` optional		Integer	Sets the debug level of the helper, printing to m_HelperLogfile.
`m_HelperDoNotQueryVBs` optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides m_HelperDoWeQuery.
`m_HelperDoNotStoreVBs` optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides m_HelperDoWeStore.
`m_HelperDoQueryVBs` optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.

Table 140. snmpHelper.SnmpHelperConfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDoStoreVBs` optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides m_HelperDoWeStore.
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: • 0: Do not use cache • 1: Use cache
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: • 0: Do not store replies in database • 1: Store replies in database
`m_HelperLogfile` optional		Text	The full path and file for the logfile of the current helper.
`m_HelperStartupTimeout`		Long64	The default helper startup timeout, that is, the maximum time to wait for a helper to start up when requested to.
`m_HelperReqTimeout`		Long64	The helper request timeout, that is, how long before each request expires.

SNMP helper database configuration

The following insert provides an example configuration of the SNMP helper database.

```
insert into SnmpHelper.SnmpHelperConfig
(
        m_HelperDbTimeout,
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200, 1200, 90, 0, 0
);
```

snmpHelper.SnmpHelperTable table

The snmpHelper.SnmpHelperTable database table configures the operation of the SNMP helper.

The schema of the snmpHelper.SnmpHelperTable database table is described in the following table.

Table 141. snmpHelper.SnmpHelperTable database table schema

Column name	Constraints	Data type	Description
`m_CommunitySuffix`		Text	The suffix to the community string.
`m_HostIp`	NOT NULL	Text	IP address of the device to interrogate.
`m_OID`	NOT NULL	Atom	Object ID for the Get request.
`m_Protocol`		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`m_RequestType`		Integer	Type of request: • 0: Get • 1: GetNext • 2: GetBulk
`m_SnmpIndex`		Atom	The index of the Get request (if it is a Get request).
`RivHelperDbTimeToDie`		Long64	How long the requested information is to live within the Helper Server.
`RivHelperRequestGetKey`	NOT NULL	Text	A key interface to the databases of the Helper Server for Get requests.
`RivHelperRequestOutput`		Atom	The response data.

Table 141. snmpHelper.SnmpHelperTable database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`RivHelperRequestReplyKey`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A key interface to the databases of the Helper Server for Reply requests.

snmpFilter database

The snmpFilter database is automatically populated with information about devices that have SNMP interface filters applied to them. You can also define dependent SNMP interface filters in this database table.

Description

The snmpFilter database is defined in the `NCHOME/etc/precision/DiscoSnmpHelperSchema.cfg` file.

snmpFilter.instances database table

The snmpFilter.instances database table is used by the SNMP Helper to hold cached data for devices for which filtered SNMP requests were made. This table is automatically populated. You must not configure inserts into this table.

Schema

The snmpFilter.instances database table schema is described in the following table:

Table 142. snmpFilter.instances database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_HostIP	not null	text	The IP address of the device to which this filter is applied.
m_FilterName	not null	text	The name of the filter from which the instance list was generated.
m_InstanceFilterTables	not null	list type text	The list of the tables to which this interface filter applies.
m_InstanceList		list type text	The list of instances for this device that match the filter.

TelnetHelper database

The TelnetHelper database defines the operation of the Telnet helper.

TelnetHelper.configuration database table

The TelnetHelper.configuration table specifies the general rules for receiving information from remote devices.

The TelnetHelper.configuration table is described in the following table.

Table 143. TelnetHelper.configuration database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_NumThreads		Integer	The number of threads to be used by the helper. If you change this value, be sure that your system is configured to allow at least this number of concurrent Telnet sessions.
m_TimeOut		Integer	The maximum time to wait for access to a device (milliseconds).
m_Retries		Integer	The number of times to retry the device.

TelnetHelper.deviceConfig database table

The TelnetHelper.deviceConfig table sets device-specific configuration options.

The TelnetHelper.deviceConfig table is described in the following table.

Table 144. TelnetHelper.deviceConfig database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_ContinueCmd		Text	The response to send to the remote device in order for it to continue the paged output. This is usually set to "y". You must take care setting this value, as some devices require a carriage return after the command and some do not. For maximum flexibility, a return is not added by default. It must be specified explicitly using a trailing Ctrl-M in the string.
m_ContinueMsg		Text	The expected prompt from the remote device between paged output; for example, "Do you want to continue". Regular expressions are valid entries.
m_IpOrSubNet		Text	The IP or fully qualified subnet address of the device corresponding to a particular configuration. If this is not specified, the configuration is used as the default subnet address.
m_NetMaskBits		Integer	The number of most significant bits in the netmask. This number must be specified if m_IpOrSubNet is specified.
m_PageLength		Integer	The output page length size. This is set to 0 by default; that is, no paging. If you set a page length size, you must also insert a value into the m_PageLengthCmd column in order to set a page length command.

Table 144. TelnetHelper.deviceConfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_PageLengthCmd		Text	The command to issue in order to set the output page length.
m_Protocol		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
m_SysObjectId optional		Text	The sysObjectID MIB variable to match for this configuration entry. The entry with the longest OID match will be the entry used. For example, if you specify a value of 1.3.6.1.4.1.9.1 then all devices with OIDs of the form 1.3.6.1.4.1.9.1.* will be matched. Cisco IOS devices have OIDs of the form 1.3.6.1.4.1.9.1.*. This field is ignored if m_IpOrSubNet is specified.
m_TransmissionDelay		Integer	This option allows you to customize the delay used by ncp_dh_telnet when transmitting data to a device. This may be useful if data loss or device issues occur when using the default transmission delay setting.

TelnetHelper.telnetHelperconfig database table

The TelnetHelper.telnetHelperconfig database table configures the operation of the Telnet Helper.

The TelnetHelper.telnetHelperconfig database table is described in the following table.

Table 145. TelnetHelper.telnetHelperconfig database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_HelperDbTimeout	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires.
m_HelperDebugLevel optional		Integer	Sets the debug level of the helper, printing to m_HelperLogfile.
m_HelperDoNotQueryVBs optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides m_HelperDoWeQuery.

Table 145. TelnetHelper.telnetHelperconfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDoNotStoreVBs` optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides m_HelperDoWeStore.
`m_HelperDoQueryVBs` optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.
`m_HelperDoStoreVBs` optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides `m_HelperDoWeStore`.
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: • 0: Do not use cache • 1: Use cache
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: • 0: Do not store replies in database • 1: Store replies in database
`m_HelperLogfile` optional		Text	The full path and file for the logfile of the current helper.
`m_HelperReqTimeout`		Long64	The helper request timeout, that is, how long before each request expires.
`m_HelperStartupTimeout`		Long64	The default helper start-up timeout, that is, the maximum time to wait for a helper to start up when requested.

Telnet helper database configuration

The following example insert gives a typical configuration of the Telnet helper database.

```
insert into TelnetHelper.TelnetHelperConfig
(
        m_HelperDbTimeout,
```

```
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200, 1200, 90, 0, 0
);
```

TelnetHelper.telnetHelperTable database table

The TelnetHelper.telnetHelperTable database table configures the operation of the Telnet helper.

The TelnetHelper.telnetHelperTable table is described in the following table.

Table 146. TelnetHelper.telnetHelperTable database table schema

Column name	Constraints	Data type	Description
`m_Protocol`		Integer	An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
`RivHelperRequestReplyKey`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A unique request reply key interface to the databases of the Helper Server.
`RivHelperRequestGetKey`	NOT NULL	Text	A request get key interface to the databases of the Helper Server.
`RivHelperDbTimeToDie`		Long64	How long the requested information is to live within the Helper Server.
`m_HostIp`	NOT NULL	Text	IP address of the device to interrogate.
`m_TelnetCommand`		Text	The Telnet command.
`RivHelperRequestOutput`		Atom	The response data.

XmlRpcHelper database

The XmlRpcHelper helper database is defined in `NCHOME/etc/precision/DiscoHelperServerSchema.cfg`.

XmlRpcHelper.configuration table

The XmlRpcHelper.configuration database table configures the threads and timeout for the XMLRPC helper.

The XmlRpcHelper.configuration database table must contain only one record. The XmlRpcHelper.configuration database table is described in the following table.

Table 147. XmlRpcHelper.configuration database table schema

Column name	Constraints	Data type	Description
`m_NumThreads`	None	Integer	The number of threads to be used by the helper.
`m_TimeOut`	None	Integer	The maximum time to wait for a reply from an EMS collector, in milliseconds. If you are running the TraceRoute agent you may need to increase this value, depending on network conditions.

XmlRpcHelper.XmlRpcHelperConfig table

The XmlRpcHelper.XmlRpcHelperConfig helper database table configures the operation of the XMLRPC Helper.

The schema of the XmlRpcHelper.XmlRpcHelperConfig database table is described in the following table.

Table 148. XmlRpcHelper.XmlRpcHelperConfig database table schema

Column name	Constraints	Data type	Description
`m_HelperDbTimeout`	UNIQUE	Long64	The helper database timeout, that is, how long before the database expires.
`m_HelperDebugLevel` optional		Integer	Sets the debug level of the helper, printing to the file specified in m_HelperLogfile.
`m_HelperDoNotQueryVBs` optional		Object type varbinds	List of helper inputs that do not query the database. This field overrides m_HelperDoWeQuery.
`m_HelperDoNotStoreVBs` optional		Object type varbinds	List of helper inputs that never store data in the Helper Server databases. This field overrides m_HelperDoWeStore.
`m_HelperDoQueryVBs` optional		Object type varbinds	List of helper inputs that always query the database before querying the network. If the item is found in the database then the network is not queried.
`m_HelperDoStoreVBs` optional		Object type varbinds	List of helper inputs that always store data in the Helper Server database. This field overrides m_HelperDoWeStore.

Table 148. XmlRpcHelper.XmlRpcHelperConfig database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_HelperDoWeQuery`		Integer	Indicates whether the Helper Server queries its database or whether it queries the network using a helper: • 0: Do not use cache • 1: Use cache Because each data item is requested from the Collector only once, caching is not usually enabled.
`m_HelperDoWeStore`		Integer	Indicates whether the Helper Server stores any replies from the helpers in its database: • 0: Do not store replies in database • 1: Store replies in database Because each data item is requested from the Collector only once, caching is not usually enabled.
`m_HelperLogFile` optional		Text	The full path and file for the logfile of the current helper.
`m_HelperReqTimeout`		Long64	The helper request timeout that is, how long before each request expires.
`m_HelperStartupTimeout`		Long64	The default helper startup timeout, that is, the maximum time to wait for a helper to start up when requested to.

XmlRpcHelper.config database configuration

The following insert provides a typical example configuration of the XmlRpcHelper database. This insert specifies the following settings:

- Helper database expires after 3 days.
- Each helper database request timeout expires after 20 minutes.
- Maximum time to wait for a helper to start up when requested is 90 seconds.
- Helper Server does not query its database.
- Helper Server does not store any replies from the helpers in its database.

```
insert into XmlRpcHelper.XmlRpcHelperConfig
(
        m_HelperDbTimeout,
        m_HelperReqTimeout,
        m_HelperStartupTimeout,
        m_HelperDoWeQuery,
        m_HelperDoWeStore
)
values
(
        259200,
 1200,
 90,
 0,
```

```
0
);
```

XmlRpcHelper.XmlRpcHelperTable table

The XmlRpcHelper.XmlRpcHelperTable configures the operation of the XMLRPC helper.

The schema of the XmlRpcHelper.XmlRpcHelperTable database table is described in the following table.

Table 149. XmlRpcHelper.XmlRpcHelperTable database table schema

Column name	Constraints	Data type	Description
`m_DataSourceId`		Integer	Data source of interest.
`m_Host`	NOT NULL	Text	The IP address of the physical device.
`m_MethodCalled`		Text	Method called.
`m_MethodSignature`		Integer	Method signature.
`m_Port`		Atom	Port of physical device.
`RivHelperDbTimeToDie`		Text	How long the requested information is to live within the Helper Server.
`RivHelperRequestGetKey`	NOT NULL	Text	A key interface to the databases of the Helper Server for Get requests.
`RivHelperRequestOutput`		Atom	Response data.
`RivHelperRequestReplyKey`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	A key interface to the databases of the Helper Server for Reply requests.

Tracking discovery databases

During the discovery process, the discovery engine, ncp_disco, records every element discovered in the network, whether it has been processed or not. The instrumentation and translations databases are used for this purpose. These databases can be interrogated at any time to view the number of device types and categories that have been discovered.

The translations, instrumentation, and workingEntities databases record the known network entities and technologies, and can be used to track the progress of the discovery.

translations database

The translations database is defined in the `$NCHOME/etc/precision/DiscoSchema.cfg` file. It has several fully qualified database table names.

The fully qualified database table names for the translations database are as follows:

- translations.ipToBaseName
- translations.vlans
- translations.NAT

- translations.NATtemp
- translations.NATAddressSpaceIds
- specialManagementIPs

translations.ipToBaseName table

The ipToBaseName table is a registry of discovered devices and the IP addresses associated with those devices.

When a device has multiple interfaces, and therefore multiple IP addresses, the Associated Address agent downloads all the associated addresses, stores them in the ipToBaseName table and allows the appropriate discovery agents to discover the device. Any subsequent attempt to discover the device by means of another of its IP addresses is halted when the Associated Address agent checks the ipToBaseName table, that is, before the device details are passed to the appropriate discovery agent.

Table 150. translations.ipToBaseName database table schema

Column name	Constraints	Data type	Description
`m_BaseName`	NOT NULL	Text	Base name of the discovered entity.
`m_BaseAddress`	NOT NULL	Text	Base address of the discovered entity.
`m_WorkAddress`	NOT NULL	Text	The address that was used for data retrieval.
`m_IpAddress`	NOT NULL	Text	IP address of the entity.
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
`m_InScope`		Boolean integer	Indicates whether the value of the field `m_IpAddress` is in scope.
`m_Protocol`	NOT NULL	Integer	Protocol for this address. This field can take the following values: • 1: *IPv4* • 3: *IPv6*
`m_IsManagementIP`		Boolean integer	Indicates whether this is a management IP address.
`m_IsOutOfBand`		Boolean integer	Indicates whether this is an out of band address.
`m_Name`		Text	Name of interface with IP if known.

translations.vlans table

The vlans table holds a list of devices that are part of Virtual Local Area Networks (VLANs). Each record in the vlans table maps the device to the VLAN to which it belongs.

Table 151. translations.vlans database table schema

Column name	Constraints	Data type	Description
m_Name	• PRIMARY KEY • NOT NULL	Text	The name of the device associated with this entry.
m_VlanID	• PRIMARY KEY • NOT NULL	Text	The VLAN identifier on the device.
m_Subnet		Text	The subnet with which the VLAN appears to be associated.
m_NetMask		Text	The subnet mask.
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

translations.NAT table

The NAT table is used to hold static NAT mappings. The mapped devices are discovered even if they are outside the scope of the discovery.

Table 152. translations.NAT database table schema

Column name	Constraints	Data type	Description
m_OutsideGlobalAddr	• PRIMARY KEY • NOT NULL	Text	The public address.
m_InsideLocalAddr	NOT NULL	Text	The private address.
m_InsideGlobalAddr		Text	This column is currently not used.
m_OutsideLocalAddr		Text	This column is currently not used.
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

translations.NATtemp

The NATtemp table is used to hold NAT mappings from a particular NAT gateway. This enables the discovery process to compare the old and new NAT mappings and initiate a partial or full rediscovery if necessary.

Table 153. translations.NATtemp database table schema

Column name	Constraints	Data type	Description
`m_OutsideAddr`	• PRIMARY KEY • NOT NULL	Text	The public address of the device.
`m_InsideAddr`	NOT NULL	Text	The private address of the device.
`m_AddressSpace`		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

translations.NATAddressSpaceIds table

The NATAddressSpaceIds table is used to identify the IP addresses of NAT gateways and specify an address-space identifier for each one.

Table 154. translations.NATAddressSpaceIds database table schema

Column name	Constraints	Data type	Description
`m_NATGatewayIP`	• PRIMARY KEY • NOT NULL	Text	The IP address of the gateway.
`m_AddressSpaceId`		Text	The address space identifier to be used for all devices in the NAT domain belonging to the gateway whose IP address is specified in m_NATGatewayIP.

specialManagementIPs table

After the discovery processing phase, this table contains an entry for each IP address that was in scope, based on the entries in the scope.special table.

Table 155. specialManagementIPs table

Column	Constraints	Data type	Description
m_IpAddress	Not null	Text	The IP address of the entity.
m_WorkAddress	Not null	Text	The address that was used for data retrieval
m_AdminInterfaceIP		Int type Boolean	Indicates whether the address is an interface, as defined in the scope.special table

Table 155. specialManagementIPs table (continued)			
Column	**Constraints**	**Data type**	**Description**
m_IsManagementIP		Int type Boolean	Indicates whether the address is a management address, as defined in the scope.special table
m_ExtraInfo		Object type VB list	The extra information that enriches the target entity, as defined in the scope.special table.
m_AddressSpace		Text	The address space for this IP as defined in the ipToBaseName table.
m_Identifier		Text	The identifier, as defined in the scope.special table.
m_Priority		Int	The priority, as defined in the scope.special table.
m_NonPingable		Int	Indicates whether the address is selected even if it cannot be pinged, as defined in the scope.special table
m_UsedForChassis		Int	If 1then this IP address was assigned to be used as the access address for a chassis entity.

instrumentation database schema

The instrumentation database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. It lists discovered devices grouped by technology. You can do OQL queries to retrieve the names of all discovered subnets, VLANs, Frame Relay devices, and so on.

The fully qualified database table names for the instrumentation database are:

- instrumentation.ipAddresses
- instrumentation.name
- instrumentation.subNet
- instrumentation.vlan
- instrumentation.frameRelay
- instrumentation.ciscoFrameRelay
- instrumentation.hsrp
- instrumentation.pnniPeerGroup
- instrumentation.fddi

instrumentation.ipAddresses table

The ipAddresses table contains a record of the unique IP addresses discovered in the network.

Table 156. instrumentation.ipAddresses database table schema

Column name	Constraints	Data type	Description
`m_UniqueAddress`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The IP address of a discovered network entity.

instrumentation.name table

The name table contains a record of the unique name of every discovered device.

Table 157. instrumentation.name database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The name of a discovered network entity.

instrumentation.subNet table

The `subNet` table contains a record of every discovered subnet address and mask.

Table 158. instrumentation.subNet database table schema

Column name	Constraints	Data type	Description
`m_SubNet`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The subnet address of a discovered subnet.
`m_NetMask`	• NOT NULL • UNIQUE	Text	The subnet mask of a discovered subnet.

instrumentation.vlan table

The vlan table contains a record of every discovered VLAN.

Table 159. instrumentation.vlan database table schema

Column name	Constraints	Data type	Description
`m_Vlan`	UNIQUE	Integer	The ID of the discovered VLAN.

instrumentation.frameRelay table

The frameRelay table contains a record of every discovered Frame Relay device.

Table 160. instrumentation.frameRelay database table schema

Column name	Constraints	Data type	Description
`m_IfDlci`	• PRIMARY KEY • NOT NULL • UNIQUE	Integer	The Frame Relay device Data Link Connection Identifier.
`m_IfIndex`	• PRIMARY KEY • NOT NULL	Integer	The unique value for each device interface.

instrumentation.ciscoFrameRelay table

The ciscoFrameRelay table contains a record of every discovered Cisco Frame Relay device.

Table 161. instrumentation.ciscoFrameRelay database table schema

Column name	Constraints	Data type	Description
`m_UniqueKey`	• NOT NULL • UNIQUE	Text	A combination of the IP Address, the FRIfIndex, and the FRDlci.
`m_FRIfIndex`	• PRIMARY KEY • NOT NULL	Integer	The unique value for each device interface.
`m_FRDlci`	• PRIMARY KEY • NOT NULL • UNIQUE	Integer	The Frame Relay device Data Link Connection Identifier.

instrumentation.hsrp table

The hsrp table contains a record of every discovered Hot Standby Router Protocol (HSRP) device.

Table 162. instrumentation.hsrp database table schema

Column name	Constraints	Data type	Description
`m_GroupAddress`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The group address of the device.
`m_PrimaryAddress`		Text	The primary address of the device.
`m_StandbyAddress`		Text	The standby address of the device.

instrumentation.pnniPeerGroup table

The pnniPeerGroup table contains the lowest level Peer Group Identifiers of PNNI devices that have been discovered. Logical PNNI Peer Groups IDs are not stored.

Table 163. instrumentation.pnniPeerGroup database table schema

Column name	Constraints	Data type	Description
m_PeerGroupId	• PRIMARY KEY • NOT NULL • UNIQUE	Text	The lowest level PNNI peer group identifier.

instrumentation.fddi table

The fddi table contains the Fibre Distributed Data Interface (FDDI) nodes that have been discovered.

Table 164. instrumentation.fddi database table schema

Column name	Constraints	Data type	Description
m_UniqueAddress	• PRIMARY KEY • NOT NULL	Text	The unique address of the node.
m_StationManagmentTask	• PRIMARY KEY • NOT NULL	Integer	The station management task for that node.

workingEntities database

The workingEntities database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. Its fully qualified database table names are: workingEntities.finalEntity; workingEntities.containment.

The workingEntities database provides a central repository for information about discovered entities and the containment details associated with each of these entities. However, this database is populated only at the end of the discovery process.

workingEntities.finalEntity table

The finalEntity table is a central repository for information about discovered entities.

Table 165. workingEntities.finalEntity database table schema

Column name	Constraints	Data type	Description
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
m_BaseName		Text	The name of the Base Entity for this device.
m_Creator	NOT NULL	Text	Name of agent (or finder) that discovered the entity.

Table 165. workingEntities.finalEntity database table schema (continued)

Column name	Constraints	Data type	Description
`m_Description`		Text	Description of the device, taken from the `sysDescr` MIB variable for the entity.
`m_EntityType`	Externally defined entityType data type	Integer	Element type description of the discovered entity: • 0: Unknown type • 1: Base entity • 2: Local neighbor • 3: Remote neighbor
`m_ExtraInfo`	Externally defined vblist data type	Object	Extra information requested by the agent.
`m_HaveAccess`	Externally defined Boolean data type	Boolean integer	Flag indicating whether SNMP access to the device is available: • 1: SNMP access is available • 0: No SNMP Access
`m_IsActive`	Externally defined Boolean data type	Boolean Integer	Indicates whether the entity is active: • (2) Indicates that the entity is discovered but is not in scope. Entities that are not in scope are not monitored by Network Manager. • (1) Entity is active. • (0) Entity is *in*active.
`m_LocalNbr`	Externally defined vblist data type	Object	Information about the local neighbor.
`m_Name`	• PRIMARY KEY • NOT NULL • UNIQUE	Text	Unique name of the discovered entity.
`m_ObjectId`		Text	Device class (a textual representation of the ASN.1 address).
`m_UniqueAddress`		Text	IP address of the network entity.

workingEntities.containment table

The containment table is a central repository for information about containment information for discovered entities. It shows the containment relationships between all entities in the finalEntity table.

As an example of how the containment table works, assume the finalEntity table includes the following distinct entities:

- A device with IP address `1.2.3.4`
- An interface on this device, `1.2.3.4[0[1]]`

The finalEntity table provides no containment information for these two entities. In other words, it does not indicate that the interface 1.2.3.4[0[1]] is physically contained within the device 1.2.3.4. This containment information is held within the containment table, as follows:

```
m_Container='1.2.3.4'
m_Part='1.2.3.4[0[1]]'
m_IsPhysical=1
m_LinkType=1
```

Note that m_Container and m_Part are each unique names of entities on the network, each with a unique m_Name in the finalEntity table.

Table 166. workingEntities.containment database table schema

Column name	Constraints	Data type	Description
`m_Container`	• PRIMARY KEY • NOT NULL	Text	The name of an object which contains something. This object refers to an entity on the network and corresponds to an entity with its own entry and unique m_Name in the workingEntities.finalEntity table.
`m_Part`	• PRIMARY KEY • NOT NULL	Text	The name of the object which is contained. This object refers to an entity on the network and corresponds to an entity with its own entry and unique m_Name in the workingEntities.finalEntity table.
`m_IsPhysical`		Boolean integer	Flag indicating whether the containment is physical or logical: • 1: Physical Containment • 0: Logical Containment
`m_LinkType`		Integer	Value indicating mode of data transfer between `m_Container` and `m_Part`. The following values are possible: • 0: No data is transmitted. • 1: Data is transmitted both ways. • 2: Data travels from m_Container to m_Part. • 3: Data travels from m_Part to m_Container

workingEntities.interfaceMapping

The interfaceMapping table enables the stitching to quickly identify interfaces.

The following table lists the columns in the interfaceMapping table.

Note: Not all the fields in this table are populated; however, the use of this table provides a fast way of looking up data.

Table 167. workingEntities.interfaceMapping database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Name	not null	Text	Unique name of an interface on the network.
m_IfIndex		Integer	SNMP ifIndex.
m_InterfaceId		Text	Interface identifier.
m_EntPhysIndex		Integer	Entity MIB physical Index if present.
m_IfDescr		Text	Interface RFC.ifDescr.
m_IfName		Text	Interface RFC ifName.
m_IfAlias		Text	Interface RFC ifAlias field.
m_IfType		Integer	Interface RFC ifType.
m_PhysAddress		Text	MAC address for this entity if present.
m_BaseName	Not null	Text	Name of the "Base Entity" for this device.
m_AddressSpace		Text	Name of the address space this device is on. For public devices the field is null.

dbModel database

The dbModel database maps custom discovery data from discovery agents to NCIM tables.

The dbModel database is used for mapping data that hs been retrieved by discovery agents to the appropriate tables in the NCIM topology database. It is defined in the `NCHOME/etc/precision/ModelNcimDB.cfg` configuration file.

dbModel.access table

The dbModel.access table configures database access.

The following table shows the schema for the dbModel.access database table.

Table 168. dbModel.access database table schema			
Column name	**Constraints**	**Data type**	**Description**
EnumGroupFilter	NOT NULL	Text	Lists the enumerations groups that contain enumerations that can be used in the entity maps defined in the dbModel.entityMap table. The enumerations are defined in the `enumerations` table in the NCIM topology database.

Table 168. dbModel.access database table schema (continued)

Column name	Constraints	Data type	Description
TransactionLength	NOT NULL	Integer	The number of SQL statements to execute within a single transaction during topology upload before committing.
WebTopDataSource	NOT NULL	Text	The name of the Webtop Datasource to use. This value can be different from the ObjectServer name.
DomainHost		Text	The hostname that Topoviz connects to. This is set in the entry for ncp_config in the `ServiceData.cfg` configuration file. This field should be left blank unless you need to overwrite this value.
DomainPort		Integer	The port that Topoviz connects to. This is set in the entry for ncp_config in the `ServiceData.cfg` configuration file. This field should be left blank unless you need to overwrite this value.

Example: Using an enumeration group filter and entity map

In the workingEntities.finalEntity table, the OSPF interface type is stored in the `m_OspfIfState` enumerated list, which is contained in the `m_ExtraInfo` field. The value of the `m_OspfIfState` field is a single integer, for example, 3. `m_OspfIfState` corresponds to `ospfIfState` in the NCIM topology database. The enumerations for `ospfIfState` are defined for the `enumGroup` `ospfIfType` in the `enumerations` table in the NCIM database, as shown in the following example output:

```
> select * from enumerations where enumGroup = 'ospfIfType';
> go
+------------+---------+-------------------+-----------------+
| ENUMGROUP  | ENUMKEY | ENUMVALUE         | ENUMDESCRIPTION |
+------------+---------+-------------------+-----------------+
| ospfIfType | 1       | broadcast         |                 |
| ospfIfType | 2       | nbma              |                 |
| ospfIfType | 3       | pointToPoint      |                 |
| ospfIfType | 5       | pointToMultipoint |                 |
+------------+---------+-------------------+-----------------+
```

The following example insert includes **ospfIfType** (shown here in bold type) in the enumerations to be downloaded:

```
insert into dbModel.access
(
    EnumGroupFilter,
    TransactionLength,
    WebTopDataSource
)
values
(
    "enumGroup in ('ASN' , 'sysServices', 'ifAdminStatus', 'ifOperStatus',
    'sysServices',  'ifType', 'ifOperStatusToOperationalStatus',
    'entPhysicalClass', 'cefcFRUPowerAdminStatus', 'cefcFRUPowerOperStatus',
    'TruthValue','TruthValueString', 'entSensorType', 'entSensorScale',
    'entSensorStatus', 'cefcModuleAdminStatus', 'cefcModuleOperStatus',
    'ipForwarding', 'cefcPowerRedundancyMode', 'EntityType', 'ospfIfState',
    'ospfIfType', 'dot3StatsDuplexStatus', 'accessProtocol', 'cdmDuplex',
    'OperationalStatusEnum')",
    500,
    "NCOMS"
);
```

The following example insert into the dbModel.entityMap database shows **m_ospfIfType** (displayed in bold type) from the workingEntities.finalEntity table being mapped to the `ospfIfType` column in the `ospfEndPoint` table in the NCIM topology database.

```
insert into dbModel.entityMap
(
    EntityFilter,
    TableName,
    FieldMap,
    StitcherDefined
)
values
(
    "m_ObjectId = 'OSPF_PROTOCOL_ENDPOINT'",
    'ospfEndPoint',
    {
        entityId         = "eval(int,  '&m_EntityId')",
        areaId           = "eval(text, '&m_ExtraInfo->m_AreaId')",
        ospfIfAdminStat  = "eval(int,  '&m_ExtraInfo->m_OspfIfAdminStat')",
        ospfIfState      = "eval(text, 'LOOKUP(&m_ExtraInfo->m_OspfIfState,
&&ospfIfState)')",
        ospfIfType  = "eval(text, 'LOOKUP(&m_ExtraInfo->m_OspfIfType,
&&ospfIfType)')",
        defaultCost      = "eval(int,  '&m_ExtraInfo->m_Cost')"
    },
    1
);
```

Because the enumeration for `ospfIfType` was downloaded, the integer value of `ospfIfType` from the workingEntities.finalEntity table is mapped to a meaningful string in the record in the NCIM topology database. For example, instead of 3, the value for the interface type is stored as `pointToPoint`.

dbModel.entityDetails table

The dbModel.entityDetails table defines extra information to be added to the EntityDetails table in the NCIM topology database.

The following table shows the schema for the dbModel.entityDetails database table.

Table 169. dbModel.entityDetails database table schema

Column name	Constraints	Data type	Description
EntityType	Primary Key	Integer	Any entities of this type will have the entityDetails field in NCIM enriched with the fields from EntityDetails. A single insert is allowed per entity type.
EntityDetails	NOT NULL	Object type vblist	A list of key-value pairs that, if they are present, are inserted into the EntityDetails table in the NCIM topology database. Use this field to set multiple values for the same entity type.

dbModel.entityMap table

The dbModel.entityMap table defines how values are mapped from the discovery `workingEntities.finalEntity` table to dNCIM and NCIM.

The `NCHOME/etc/precision/ModelNcimDB.cfg` configuration file contains example inserts showing how to add data for new and existing entities.

The following table shows the schema for the dbModel.entityMap database table.

Table 170. dbModel.entityMap database table schema			
Column name	**Constraints**	**Data type**	**Description**
EntityFilter	NOT NULL	Text	The filter to apply to the contents of the workingEntities.finalEntity database table. Any entity matching the filter has the entityMap applied to it.
TableName	NOT NULL	Text	The name of table to be populated in dNCIM or NCIM.
DisplayLabel		Text	Evaluated in the same way as FieldMap. Populates the entityData.displayLabel field for different types of entity.
FieldMap	NOT NULL	Object of type vblist	Maps the fields of the table defined by the *TableName* value to the evaluation against the data from the workingEntities.finalEntity database table.
Connection		Object of type vblist	This field is not used.
Relationships		Object of type vblist	List of relationships that this entity has with other entities. For example, if it connects or contains other entities.
Iterators		Object of type vblist	Used to iterate over lists in the OQL record.
ImplicitEntities		Object of type vblist	Defines any new entities to be created in NCIM that are not explicitly represented in ncp_model.
StitcherDefined		Boolean integer	If this is 1, then the mapping is done by the stitchers and not by this table. By default, this value is 0.

Working topology databases

The discovery engine, ncp_disco, uses a series of databases to perform the data processing stages of the discovery cycle. Stitchers operate on these databases to knit together a network topology and create the containment model.

The stitchers produce the various network topologies, such as layer 2 and layer 3 topologies, by amalgamating the information in the discovery agents returns tables into a single cumulative topology within the fullTopology database.

fullTopology database schema

The fullTopology database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. Its fully qualified database table name is fullTopology.entityByNeighbor.

The fullTopology database holds the generated topology. On completion of the data collection phase of the discovery, the stitchers merge the information that has been retrieved by the discovery agents to form a single topology, which at this stage is in a name-to-name format.

fullTopology.entityByNeighbor table

The entityByNeighbor table contains information about connectivity between discovered devices.

Table 171. fullTopology.entityByNeighbor database table schema

Column name	Constraints	Data type	Description
`m_Name`	• PRIMARY KEY • NOT NULL	Text	Unique name of an entity on the network.
`m_NbrName`	• PRIMARY KEY • NOT NULL	Text	The name of the device that is connected to the unique network entity.
`m_NbrType`	Externally defined connectionType data type	Integer	Integer representation of the type of connection between the network entity and its neighbor: • 2: Main-to-Local • 3: Local-to-Remote

dNCIM schema

The dNCIM database holds the containment model that is derived from the workingEntities.finalEntity, workingEntities.containment and layer tables, mainly fullTopology.entityByNeighbor. The model is built by the stitchers located in the dNCIM subdirectory, $NCHOME/precision/disco/stitchers/DNCIM. This is the version of the topology that is sent to the ncp_model component

The dNCIM database contains the same tables as the NCIM topology database. These NCIM tables hold topology information about the network In addition, dNCIM contains extra tables that store processing data as the topology is being built.

Related concepts

Data schema
In the NCIM database, Network Manager topology data falls into different categories.

Data dictionary
The NCIM topology database schema is made up of a set of relational database tables that represent the topology model.

rediscoveryStore database

The rediscoveryStore database is used for comparison purposes in rediscovery mode. It is defined in $NCHOME/etc/precision/ DiscoSchema.cfg. Its fully qualified database table names are: rediscoveryStore.dataLibrary; rediscoveryStore.rediscoveredEntities

The rediscoveryStore database holds information from previous discovery cycles that can be used for comparison purposes during a full or partial rediscovery.

rediscoveryStore.dataLibrary table

The dataLibrary table is used as a reference point during rediscovery mode to compare the previous and present states.

Table 172. rediscoveryStore.dataLibrary database table schema

Column name	Constraints	Data type	Description
m_Name		Text	Unique name of an entity on the network.
m_UniqueAddress		Text	The IP address of a discovered network entity.
m_CompareDb	NOT NULL	Text	The entity that is used to compare this network entity.

rediscoveryStore.rediscoveredEntities table

The rediscoveredEntites table stores entities found during a rediscovery.

Table 173. rediscoveryStore.rediscoveredEntities database table schema

Column name	Constraints	Datatype	Description
m_Name		Text	Unique name of an entity on the network.
m_UniqueAddress		Text	The IP address of a discovered network entity.
m_PhysAddr		Text	The physical address of the entity.
m_OldBaseName			The base name of the entity prior to rediscovery
m_NewBaseName			The base name of the entity after rediscovery.

Topology manager databases

On completion of a discovery, ncp_model receives topology updates from dNCIM, and based on these updates, generates the necessary inserts to update the NCIM database. The Topology Manager also broadcasts these changes to other processes in Network Manager.

ncimCache database

This database stores topology updates from DNCIM.

The ncimCache database is created by ncp_model. The ncp_model process sends topology updates on the message bus to all subscribers in the format of this database. The ncp_g_event and ncp_store processes subscribe to topology updates and keep a copy of the ncimCache database.

The Event Gateway stitchers use data from the ncimCache database.

You can query the ncimCache database tables on the service `model`.

Related reference

NCIM cache files

Topology updates are held in a set of files called the NCIM cache files.

ncimCache.collects table

The ncimCache.collects table lists all the entities participating in a given collection.

The following table shows the schema for the ncimCache.collects database table.

Table 174. ncimCache.collects database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to collectingEntityID in the `collects` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the collecting entity.
MSGTYPE		String	The name of the table within the ncimCache database.
collects	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. • ENTITYNAME corresponds to collectedEntityID in the `collects` table in the NCIM database. • SEQUENCE corresponds to sequence in the `collects` table in the NCIM database.

Format of the data in the ncimCache.collects database table

The following example shows the format of the data in the ncimCache.collects database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.collects.`*`DOMAIN`* file.

```
{
    ENTITYID=31051;
    ENTITYNAME='SUBNET_OBJECT / 192.168.232.24 / 30 /';
    MSGTYPE='collects';
    collects=[
    {
        ENTITYNAME='some-device.1[ 0 [ 33 ] ]';
        SEQUENCE=0;
    },
    {
        ENTITYNAME='some-device.2[ 0 [ 25 ] ]';
        SEQUENCE=0;
    }
    ];
}
```

Related reference

collects

The `collects` table stores data on collections of entities, such as subnets and MPLS VPNs. This table belongs to the category *collections*.

ncimCache.connectActions table

The ncimCache.connectActions table lists all changes made manually to connections in the topology.

The following table shows the schema for the ncimCache.connectActions database table.

Table 175. ncimCache.connectActions database table schema			
Column name	**Constraints**	**Data type**	**Description**
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to the aEndEntityId in the `connectActions` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
MANUAL		Boolean	If the connection was added manually to the topology, this value is present and set to 1.
MSGTYPE		String	The name of the table within the ncimCache database.
connectActions	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. All names correspond to the column names in the `connectActions` table in the NCIM database.

Format of the data in the ncimCache.connectActions database table

The following example shows the format of the data in the ncimCache.connectActions database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.connectActions.`*`DOMAIN`* file.

```
{
        ENTITYID=60857;
        ENTITYNAME='MyManualDevice';
        MANUAL=1;
        MSGTYPE='connectActions';
        connectActions=[
        {
                ACTION='add';
                AENDENTITYNAME='MyManualDevice';
                CHANGETIME='2013-07-08 11:50:58';
                CONNECTACTIONSID=61;
                DESCRIPTION='';
                LOCATION='192.168.78.108';
                MANUAL=1;
                TOPOENTITYNAME='Layer1Topology';
                UNIDIRECTIONAL=0;
                USERNAME='defaultWIMFileBasedRealm/itnmadmin';
                ZENDENTITYNAME='bungle';
        }
        ];
}
```

Related reference

connectActions

The connectActions table records all manual connection additions and all connection removals, including removal of connections that were discovered rather than manually added.

ncimCache. connectstable

The ncimCache.connects table describes the type and speed of connections between devices.

The following table shows the schema for the ncimCache.connects database table.

Table 176. ncimCache.connects database table schema			
Column name	**Constraints**	**Data type**	**Description**
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to the aEndEntityId in the `connects` table in the NCIM database.

Table 176. ncimCache.connects database table schema (continued)

Column name	Constraints	Data type	Description
ENTITYNAME	NOT NULL	String	The name of the aEndEntityId device.
MANUAL		Boolean	If the connection was added manually to the topology, this value is present and set to 1.
MSGTYPE		String	The name of the table within the ncimCache database.
connectSpeeds	NOT NULL	List of name/value pairs	A list of name/value pairs for the connection. **ENTITYNAME** The name of the zEndEntityId device in the `connects` table in the NCIM database. **SPEEDTYPE** Corresponds to `speedType` in the `connectSpeeds` table in the NCIM database. **SPEEDVALUE** Corresponds to `speedValue` in the `connectSpeeds` table in the NCIM database. **UNIDIRECTIONAL** Corresponds to `unidirectional` in the `connects` table in the NCIM database.
connects	NOT NULL	List of name/value pairs	A list of name/value pairs for the connection. **ENTITYNAME** Corresponds to `zEndEntityID` in the `connects` table in the NCIM database. **MANUAL** This value is 1 if the connection was manually added. The connection can be between discovered devices, manually added devices, or both. If the connection was not manually added, this name/value pair is not present. **TOPOENTITYNAME** Corresponds to `entityId` in the `topologyLinks` table in the NCIM database. **UNIDIRECTIONAL** Corresponds to `unidirectional` in the `connects` table in the NCIM database.

Format of the data in the ncimCache.connects database table

The following example shows the format of the data in the ncimCache.connects database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.connects.`*`DOMAIN`* file.

```
{
      ENTITYID=42795;
      ENTITYNAME='mydevice[ 0 [ 25 ] ]';
      MSGTYPE='connects';
```

```
      connectSpeeds=[
      {
            ENTITYNAME=''mydevice[ 0 [ 33 ] ]';
            SPEEDTYPE='DEFAULT';
            SPEEDVALUE=100000000;
            UNIDIRECTIONAL=0;
      }
      ];
      connects=[
      {
            ENTITYNAME=''mydevice[ 0 [ 33 ] ]';
            TOPOENTITYNAME='IpPathTopology';
            UNIDIRECTIONAL=0;
      },
      {
            ENTITYNAME=''mydevice[ 0 [ 33 ] ]';
            TOPOENTITYNAME='RouterLinksTopology';
            UNIDIRECTIONAL=0;
      },
      {
            ENTITYNAME=''mydevice[ 0 [ 33 ] ]';
            TOPOENTITYNAME='RelatedToTopology';
            UNIDIRECTIONAL=0;
      },
      {
            ENTITYNAME=''mydevice[ 0 [ 33 ] ]';
            TOPOENTITYNAME='ConvergedTopology';
            UNIDIRECTIONAL=0;
      }
      ];
}
```

Related reference

connects
The connects table stores data on connectivity between devices. This table belongs to the category *collections*.

connectSpeeds
The connectSpeeds table stores data on connectivity speed between devices.

ncimCache. containstable

The ncimCache.contains table lists containment information for a device.

The following table shows the schema for the ncimCache.contains database table.

Table 177. ncimCache.contains database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to containingEntityID in the `contains` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the collecting entity.
MSGTYPE		String	The name of the table within the ncimCache database.
contains	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. • ENTITYNAME is the collectedEntityID in the contains table in the NCIM database. • UPWARDCONNECTION corresponds to upwardConnection in the `contains` table in the NCIM database.

Format of the data in the ncimCache.contains database table

The following example shows the format of the data in the ncimCache.contains database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.contains.*DOMAIN* file.

```
{
    ENTITYID=40892;
    ENTITYNAME='my-device.mylab';
    MSGTYPE='contains';
    contains=[
    {
        ENTITYNAME='my-device.mylab[ 0 [ 31 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 19 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 20 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 22 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 23 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 24 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 25 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 26 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 27 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 28 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 30 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 33 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 35 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 37 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 39 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 41 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 43 ] ]';
        UPWARDCONNECTION=1;
    },
    {
```

```
        ENTITYNAME='my-device.mylab[ 0 [ 45 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 46 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 47 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 21 ] ]';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab_SLOT_I2_R0';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab_SLOT_I12_R1';
        UPWARDCONNECTION=1;
    },
    {
        ENTITYNAME='my-device.mylab_SLOT_I32_R2';
        UPWARDCONNECTION=1;
    }
    ];
}
```

Related reference

contains

The `contains` table stores data on physical and logical containment. This table belongs to the category *containment*.

ncimCache.dependency table

The ncimCache.dependency table lists entities that are dependent on other devices.

The following table shows the schema for the ncimCache.dependency database table.

Table 178. ncimCache.dependency database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to the independentEntityID in the `dependency` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the independent entity.
MSGTYPE		String	The name of the table within the ncimCache database.
dependency	NOT NULL	List of name/value pairs	A list of name/value pairs for dependent entities. • DEPENDENCYTYPE corresponds to dependencyType in the dependency table in the NCIM database. • ENTITYNAME corresponds to dependentEntityID in the dependency table in the NCIM database.

Format of the data in the ncimCache.dependency database table

The following example shows the format of the data in the ncimCache.dependency database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.dependency.`*`DOMAIN`* file.

In this example, the cell depends on the base station.

```
{
        ENTITYID=60844;
        ENTITYNAME='baseStation12';
        MSGTYPE='dependency';
        dependency=[
        {
                DEPENDENCYTYPE=0;
                ENTITYNAME='Cell-01';
        }
        ];
}
```

Related reference

dependency

The dependency table defines a general dependency between two entities. This table belongs to the category *dependency*.

ncimCache.domainMembers table

The ncimCache.domainMembers table shows the domain to which an entity belongs.

The following table shows the schema for the ncimCache.domainMembers database table.

Table 179. ncimCache.domainMembers database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to entityId in the `domainMembers` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
MANUAL		Boolean	If the entity was added manually to the topology, this value is present and set to 1.
MSGTYPE		String	The name of the table within the domainMembers database.
domainMembers	NOT NULL	List of name/value pairs	A list of name/value pairs for the entity. • DOMAINNAME corresponds to domainMgrId in the `domainMembers` table in the NCIM database.

Format of the data in the ncimCache.domainMembers database table

The following example shows the format of the data in the ncimCache.domainMembers database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.domainMembers.`*`DOMAIN`* file.

```
{
    ENTITYID=42739;
    ENTITYNAME='my-device.mylab[ 0 [ 33 ] ]';
    MSGTYPE='domainMembers';
    domainMembers=[
    {
        DOMAINNAME='DOMAIN1';
    }
    ];
}
```

Related reference

domainMembers

The domainMembers table stores information on membership of entities within domains. This table belongs to the category *domains*.

ncimCache.entityActions table

The ncimCache.entityActions table lists all devices added using the manual topology API.

The following table shows the schema for the ncimCache.entityActions database table.

Table 180. ncimCache.entityActions database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to entityID in the entityActions table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
MSGTYPE		String	The name of the table within the ncimCache database.
MANUAL		Boolean	This value is always 1, showing that the device was added manually to the topology.
entityActions	NOT NULL	List of name/value pairs	A list of name/value pairs for the entity. All names correspond to the column names in the entityActions table in the NCIM database.

Format of the data in the ncimCache.entityActions database table

The following example shows the format of the data in the ncimCache.entityActions database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.entityActions.*DOMAIN* file.

```
{
        ENTITYID=60857;
        ENTITYNAME='MyManualDevice';
        MANUAL=1;
        MSGTYPE='entityActions';
        entityActions={
                ACTION='add';
                CHANGETIME='2013-07-08 11:50:31';
                DESCRIPTION='';
                DOMAINNAME='STEPH';
                ENTITYACTIONSID=106;
                ENTITYID=60857;
                ENTITYNAME='MyManualDevice';
                LOCATION='192.168.78.108';
                MANUAL=1;
                USERNAME='defaultWIMFileBasedRealm/itnmadmin';
        };
}
```

Related reference

entityActions

The entityActions table records all manual node additions and all node removals, including removal of nodes that were discovered rather than manually added and the swapping of nodes into and out of a domain.

ncimCache.entityData table

The ncimCache.entityData table holds different kinds of data about entities.

The following table shows the schema for the ncimCache.entityData database table.

Table 181. ncimCache.entityData database table schema			
Column name	**Constraints**	**Data type**	**Description**
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to the entityId in the `entityData` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
MANUAL		Boolean	If the entity was added manually to the topology, this value is present and set to 1.
MSGTYPE		String	The name of the table within the ncimCache database.
entityData	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity, corresponding to the entityData table in NCIM. This section contains a separate row per entity in the entityData table. The information that is included here depends on the type of the entity. For example, a chassis has different information available than an interface.
Other table names		List of name/ value pairs	The lists of name/value pairs depend on the information that is available for the entity. The name of the list corresponds to the equivalent database table in NCIM.

Format of the data in the ncimCache.entityData database table for a chassis

The following example shows the format of the data in the ncimCache.entityData database table, as shown either using an OQL query, or as seen in the NCHOME/`var/precision/Store.Cache.ncimCache.entityData.`*`DOMAIN`* file.

This example shows data for a chassis, corresponding to the classMembers, computerSystem, discoverySource, entityData, operatingSystem, physicalChassis, and snmpSystem tables in NCIM.

```
{
    BASENAME='xx-xx-xxnn.xx.test.lab';
    ENTITYID=40892;
    ENTITYNAME='xx-xx-xxnn.xx.test.lab';
    ENTITYTYPE=1;
    METACLASS='Element';
    MSGTYPE='entityData';
    classMembers={
        CLASSID=33;
        ENTITYID=99999;
    };
    computerSystem={
        ENTITYID=99999;
    };
    discoverySource=[
    {
        DISCOVERYPROTOCOL='SNMP';
        ENTITYID=40892;
        MANAGEDBY='DirectAccess';
        SOURCE='Agent';
    }
    ];
    entityData={
        CDMADMINSTATE=0;
        CHANGETIME='2013-06-26 14:21:10';
        CREATETIME='2013-06-26 14:21:10';
        DESCRIPTION='Cisco IOS Software, 2800 Software (C2800NM-ADVIPSERVICESK9-M),
Version 12.4(24)T7, RELEASE SOFTWARE (fc2)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2012 by Cisco Systems, Inc.
Compiled Tue 28-Feb-12 10:43 by prod_rel_team';
```

```
        DISPLAYLABEL='xx-xx-xxnn.xx.test.lab';
        ENTITYID=40892;
        ENTITYNAME='xx-xx-xxnn.xx.test.lab';
        ENTITYTYPE=1;
        MAINNODEENTITYID=40892;
        MANUAL=0;
    };
    operatingSystem={
        ENTITYID=40892;
    };
    physicalChassis={
        ACCESSIPADDRESS='192.168.233.103';
        ACCESSPROTOCOL='IPv6';
        CDMTYPE=2;
        CLASSNAME='Cisco28xx';
        ENTITYID=40892;
        FWREVISION='System Bootstrap, Version 12.4(1r) [hqluong 1r],
RELEASE SOFTWARE (fc1)';
        HWREVISION='V02 ';
        INTERFACECOUNT=47;
        ISIPFORWARDING='forwarding';
        MANUFACTURER='Cisco';
        MODEL='CISCO2811      ';
        NAME='2811 chassis';
        PARTNUMBER='CISCO2811      ';
        PHYSICALINDEX=1;
        RELATIVEPOSITION=-1;
        SERIALNUMBER='XXXXXXXXXXX';
        SERVICES='datalink(2) network(3) transport(4) application(7)';
        SWREVISION='12.4(24)T7, RELEASE SOFTWARE (fc2)';
        VENDORTYPE='1.3.6.1.4.1.9.12.3.1.3.436';
    };
    snmpSystem={
        ENTITYID=40892;
        SYSCONTACT='example@example.com';
        SYSDESCR='Cisco IOS Software, 2800 Software (C2800NM-ADVIPSERVICESK9-M),
Version 12.4(24)T7, RELEASE SOFTWARE (fc2)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2012 by Cisco Systems, Inc.
Compiled Tue 28-Feb-12 10:43 by prod_rel_team';
        SYSLOCATION='B510/3D32 3rd Floor Lab';
        SYSNAME='xx-xx-xxnn.xx.test.lab';
        SYSOBJECTID='1.3.6.1.4.1.9.1.576';
    };
}
```

Format of the data in the ncimCache.entityData database table for an interface

The following example shows the format of the data in the ncimCache.entityData database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.entityData.*DOMAIN* file.

This example shows data for a network interface, corresponding to the entityData, networkInterface, and physicalConnector tables in NCIM.

```
{
    BASENAME='my-device1.mylab';
    CONNECTIONS=['my-device2.mylabb[ 0 [ 25 ] ]'];
    ENTITYID=42739;
    ENTITYNAME='my-device1.mylab[ 0 [ 33 ] ]';
    ENTITYTYPE=2;
    METACLASS='Element';
    MSGTYPE='entityData';
    entityData={
        CDMADMINSTATE=2;
        CHANGETIME='2013-06-26 14:21:24';
        CREATETIME='2013-06-26 14:21:24';
        DISPLAYLABEL='[ IfIndex:33 ]';
        ENTITYID=42739;
        ENTITYNAME='my-device1.mylab[ 0 [ 33 ] ]';
        ENTITYTYPE=2;
        MAINNODEENTITYID=40892;
        MANUAL=0;
    };
    networkInterface={
        ACCESSIPADDRESS='2222:22a:2a2e:222::22';
        ACCESSPROTOCOL='IPv6';
        CONNECTORPRESENT='false';
```

```
        ENTITYID=42739;
        IFADMINSTATUS='up';
        IFALIAS='to my-device';
        IFDESCR='Vlan25';
        IFHIGHSPEED=100;
        IFINDEX=33;
        IFNAME='Vl25';
        IFOPERSTATUS='up';
        IFSPEED=100000000;
        IFTYPE=53;
        IFTYPESTRING='propVirtual';
        MTU=1500;
        OPERATIONALDUPLEX='Unknown';
        OPERATIONALSTATUS='started';
        PHYSICALADDRESS='00:22:22:22:22:22';
        PROMISCUOUS='false';
    };
    physicalConnector={
        CDMTYPE=9;
        ENTITYID=42739;
    };
}
```

Format of the data in the ncimCache.entityData database table for a connection

The following example shows the format of the data in the ncimCache.entityData database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.entityData.*DOMAIN* file.

This example shows data for an IP connection, corresponding to the entityData and ipConnection tables in NCIM.

```
{
    BASENAME='my-device1.mylab[ 0 [ 33 ] ]->my-device2.mylab[ 0 [ 25 ] ]';
    ENTITYID=50401;
    ENTITYNAME='my-device1.mylab[ 0 [ 33 ] ]->my-device2.mylab[ 0 [ 25 ] ]';
    ENTITYTYPE=40;
    METACLASS='Element';
    MSGTYPE='entityData';
    entityData={
        CDMADMINSTATE=0;
        CHANGETIME='2013-07-03 10:46:35';
        CREATETIME='2013-07-03 10:46:35';
        DESCRIPTION='Sequential Hop between my-device1.mylab[ 0 [ 33 ] ] and
my-device2.mylab[ 0 [ 25 ] ]';
        DISPLAYLABEL='my-device.mylab[ 0 [ 33 ] ]->my-device2.mylab[ 0 [ 25 ] ]';
        ENTITYID=50401;
        ENTITYNAME='my-device1.mylab[ 0 [ 33 ] ]->my-device2.mylab[ 0 [ 25 ] ]';
        ENTITYTYPE=40;
        MANUAL=0;
    };
    ipConnection={
        ENTITYID=50401;
    };
}
```

Related reference

entityData

The entityData table stores data on entities. This table belongs to the category *entities*.

ncimCache.hostedService table

The ncimCache.hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

The following table shows the schema for the ncimCache.hostedService database table.

Table 182. ncimCache.hostedService database table schema			
Column name	**Constraints**	**Data type**	**Description**
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to hostingEntityId in the `hostedService` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the hosting entity.
MSGTYPE		String	The name of the table within the ncimCache database.
hostedService	NOT NULL	List of name/ value pairs	A list of name/value pairs for the hosted entity. • ENTITYNAME corresponds to hostedEntityID in the hostedService table in the NCIM database.

Format of the data in the ncimCache.hostedService database table

The following example shows the format of the data in the ncimCache.hostedService database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.hostedService.`*`DOMAIN`* file.

```
{
        ENTITYID=8734;
        ENTITYNAME='router3.ibm.com';
        MSGTYPE='hostedService';
        hostedService=[
        {
                ENTITYNAME='OSPF_RoutingService_ID_192.168.34.21_RD_[0]';
        }
        ];
}
```

Related reference

hostedService

A *hosted service* is a service or application running on a specific main node device. The hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

ncimCache.lingerTime table

The ncimCache.lingerTime table stores the linger time for a device.

The following table shows the schema for the ncimCache.lingerTime database table.

Table 183. ncimCache.lingerTime database table schema			
Column name	**Constraints**	**Data type**	**Description**
BASENAME			The base name of this entity.
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to entityId in the `lingerTime` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
ENTITYTYPE			The type of the entity, as enumerated in the entityType NCIM table.
MSGTYPE		String	The name of the table within the ncimCache database.

Table 183. ncimCache.lingerTime database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
lingerTime	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. **LINGERTIME** The linger time is the number of discoveries that a device can fail to be found in before it is removed from the topology. The linger time is set for a device when it is instantiated, from the default value in the model.config table. Each time that a device in the topology is not discovered, the linger time is decreased by 1. When the linger time is zero, if the device is not discovered, it is removed from the topology.

Format of the data in the ncimCache.lingerTime database table

The following example shows the format of the data in the ncimCache.lingerTime database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.lingerTime.*DOMAIN* file.

```
{
    BASENAME='somedevice.mylab';
    ENTITYID=40892;
    ENTITYNAME='somedevice.mylab';
    ENTITYTYPE=1;
    MSGTYPE='lingerTime';
    lingerTime={
        LINGERTIME=2;
    };
}
```

Related reference

lingerTime

The lingerTime table stores the linger time for a device. The linger time is the number of discoveries that a device can fail to be found in before it is removed from the topology.

ncimCache.managedStatus table

The ncimCache.managedStatus table stores the managed status information for network entities.

The following table shows the schema for the ncimCache.managedStatus database table.

Table 184. ncimCache.managedStatus database table schema			
Column name	**Constraints**	**Data type**	**Description**
BASENAME			The name of the base entity for this device.
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to entityID in the managedStatus table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the entity.
ENTITYTYPE			The type of the entity.
MANUAL		Boolean	If the entity was added manually to the topology, this value is present and set to 1.
MSGTYPE		String	The name of the table within the ncimCache database.

Table 184. ncimCache.managedStatus database table schema (continued)

Column name	Constraints	Data type	Description
managedStatus	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. **STATUS** The managed status of an entity can be one of the following values: **0** Managed state. The entity is managed. A device can be set to managed by using the Topoviz or the Structure Browser GUIs, or by using the `ManagedNode.pl` or `RemoveNode.pl` scripts. **1** Unmanaged state. The entity is unmanaged. A device can be set to unmanaged by using the Topoviz or the Structure Browser GUIs, or by using the `UnManagedNode.pl` or `RemoveNode.pl` scripts. **2** Unmanaged by ncp_disco. This setting cannot be modified from the GUI. This value is set by the `PopulateDNCIM_ManagedStatus.stch` stitcher. **3** Unmanaged because the IP address is out of the discovery scope. The device has been discovered through another IP address that is within the discovery scope. A managed status of 3 is usually given to interfaces, rather than chassis. This value is set by the `PopulateDNCIM_ManagedStatus.stch` stitcher. **Note:** Unmanaged entities do not suppress other events in RCA. The ncp_poller process does not poll unmanaged entities. Events on unmanaged entities have the field NmosManagedStatus set in the alerts.status field in the ObjectServer.

Format of the data in the ncimCache.managedStatus database table

The following example shows the format of the data in the ncimCache.managedStatus database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.managedStatus.`*`DOMAIN`* file.

```
{
    BASENAME='somedevice';
    ENTITYID=42766;
    ENTITYNAME='somedevice[ 0 [ 47 ] ]';
    ENTITYTYPE=2;
    MSGTYPE='managedStatus';
    managedStatus={
        STATUS=1;
    };
}
```

Related reference

managedStatus

The managedStatus table stores the managed status information for each network entity in the topology.

ncimCache.networkPipe table

The ncimCache.networkPipe table represents managed connections.

The following table shows the schema for the ncimCache.networkPipe database table.

Table 185. ncimCache.networkPipe database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to collectingEntityID in the networkPipe table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the collecting network pipe.
MSGTYPE		String	The name of the table within the ncimCache database.
networkPipe	NOT NULL	List of name/value pairs	A list of name/value pairs for the entity. • AENDENTITYNAME corresponds to entityId in the networkPipe table in the NCIM database. • AGGREGATIONTYPE corresponds to aggregationType in the networkPipe table in the NCIM database. • UNIDIRECTIONAL corresponds to unidirectional in the connects table in the NCIM database. • ZENDENTITYNAME corresponds to zEndEntityId in the connects table in the NCIM database.

Format of the data in the ncimCache.networkPipe database table

The following example shows the format of the data in the ncimCache.networkPipe database table, as shown either using an OQL query, or as seen in the NCHOME/var/precision/Store.Cache.ncimCache.networkPipe.*DOMAIN* file.

```
{
    ENTITYID=50401;
    ENTITYNAME='my-device.mylab[ 0 [ 33 ] ]->my-device2.lab[ 0 [ 25 ] ]';
    MSGTYPE='networkPipe';
    networkPipe=[
    {
        AENDENTITYNAME='my-device2.lab[ 0 [ 25 ] ]';
        AGGREGATIONTYPE=4;
        UNIDIRECTIONAL=0;
        ZENDENTITYNAME='my-device.mylab[ 0 [ 33 ] ]';
    }
    ];
}
```

Related reference

networkPipe

The networkPipe table represents managed connections in the network. This table belongs to the category *connectivity*.

ncimCache.pipeComposition table

The ncimCache.pipeComposition table can be used with the networkPipe table to represent a hierarchy of connections.

The following table shows the schema for the ncimCache.pipeComposition database table.

Table 186. ncimCache.pipeComposition database table schema

Column name	Constraints	Data type	Description
ENTITYID	NOT NULL	Integer	The identifier of the containing network pipe. Corresponds to groupComponent in the `pipeComposition` table in the NCIM database.
ENTITYNAME	NOT NULL	String	The name of the network pipe.
MSGTYPE		String	The name of the table within the ncimCache database.
pipeComposition	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. • AGGREGATIONSEQUENCE corresponds to `aggregationSequence` in the `collects` table in the NCIM database. • ENTITYNAME is the component network pipe, and corresponds to `partComponent` in the `pipeComposition` table in the NCIM database.

Format of the data in the ncimCache.pipeComposition database table

The following example shows the format of the data in the ncimCache.pipeComposition database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.pipeComposition.`*`DOMAIN`* file.

```
{
    ENTITYID=50402;
    ENTITYNAME='IP_Path_[172.30.233.103]->[172.30.233.101]';
    MSGTYPE='pipeComposition';
    pipeComposition=[
    {
        AGGREGATIONSEQUENCE=1;
        ENTITYNAME='my-device.mylab[ 0 [ 33 ] ]->ny-p1-cr28.na.test.lab[ 0 [ 25 ] ]';
    }
    ];
}
```

Related reference

pipeComposition
The pipeComposition table allows a higher-level connection to be defined in terms of its lower-level connections. This table belongs to the category *connectivity*.

ncimCache.protocolEndpoint table

The ncimCache.protocolEndpoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

The following table shows the schema for the ncimCache.protocolEndpoint database table.

Table 187. ncimCache.protocolEndpoint database table schema			
Column name	**Constraints**	**Data type**	**Description**
ENTITYID	NOT NULL	Integer	The identifier of an entity. Corresponds to endPointEntityID in the `protocolEndpoint` table in the NCIM database. This entity specifies protocol-specific addressing information for this endpoint.
ENTITYNAME	NOT NULL	String	The name of the end point entity.
MANUAL		Boolean	If the entity was added manually to the topology, this value is present and set to `1`.
MSGTYPE		String	The name of the table within the ncimCache database.
protocolEndpoint	NOT NULL	List of name/ value pairs	A list of name/value pairs for the entity. • ENTITYNAME corresponds to the name of the implementingEntityID entity in the protocolEndpoint table in the NCIM database. This entity implements this protocol end point. This is usually a device interface.

Format of the data in the ncimCache.protocolEndpoint database table

The following example shows the format of the data in the ncimCache.protocolEndpoint database table, as shown either using an OQL query, or as seen in the `NCHOME/var/precision/Store.Cache.ncimCache.protocolEndpoint.`*`DOMAIN`* file.

```
{
    ENTITYID=42739;
    ENTITYNAME='my-device.mylab[ 0 [ 33 ] ]';
    MSGTYPE='protocolEndPoint';
    protocolEndPoint=[
    {
        ENTITYNAME='my-device.mylab[ 0 [ 33 ] ] IP: 2222:22a:2a2e:222::22';
    },
    {
        ENTITYNAME='my-device.mylab[ 0 [ 33 ] ] IP: 192.168.222.22';
    }
    ];
}
```

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

model database schema

This database stores information about the topology so that during rediscovery, topologies can be merged efficiently.

The model database is defined in `NCHOME/etc/precision/ ModelSchema.cfg`. Its fully qualified database table names are: model.config; model.profilingData, and model.statistics.

model.config table

The model.config table stores the configuration information that is used by MODEL during rediscovery.

Table 188. model.config database table schema

Column name	Constraints	Data type	Description
ChassisCreation Events	NOT NULL	Boolean Integer	If set to 1, generates ItnmEntityCreation and ItnmEntityDeletion events when a chassis entity is created.
DiscoveryUpdateMode	NOT NULL	Integer	For internal system use only; do not modify. Prior to a batch update, **ncp_disco** sets this value to 1 for a partial discovery, or to 0 for a full discovery.
DeleteRenamedDevices	NOT NULL	Boolean Integer	Controls whether duplicate nodes are created in the topology if a device name, that is, the EntityName, is changed between discovery cycles but the IP address of the device remains the same. Possible values are as follows. An example describes the behavior depending on the value. In this example, the topology contains a node that is called deviceA.home.com, which has a LingerTime value of 3. Before the next discovery cycle, the deviceA.home.com device is renamed to deviceB.home.com. If you change this setting, restart the product for the change to take effect. • 0 (default): A duplicate node is created. The LingerTime of the existing node is decremented. In the example, the nodes deviceA.home.com and deviceB.home.com are duplicates. The LingerTime of deviceA.home.com is decremented to 2. The LingerTime of deviceB.home.com is set to 3. • 1: The existing node is overwritten by a node that has the new name of the device. In the example, the deviceA.home.com node is overwritten. A node is created for deviceB.home.com. **Important:** If you set this field to 1, you must disable sysName naming in the advanced discovery parameters.
IpInterfaceCreation Events	NOT NULL	Boolean Integer	If set to 1, generates ItnmEntityCreation and ItnmEntityDeletion events when an interface with its own IP address is created.

Table 188. model.config database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
KeepOldEntityDetails	NOT NULL	Integer	If this value is set to 0 (the default), any custom data that was added to the dbModel.entityDetails table is kept up-to-date by future discoveries. Custom data that was added but is no longer present in a discovery is removed from the NCIM topology database. If this value is set to 1, then custom data is always kept in future discoveries, unless it is manually deleted.
LingerTime	• PRIMARY KEY • NOT NULL • UNIQUE	Integer	The LingerTime value is how many discovery cycles a device can fail to be found in before it is considered as no longer present in the topology and removed.
MaintenanceState Events	NOT NULL	Boolean Integer	If set to 1, generates ItnmMaintenanceState events when the status of an entity changes in the managedStatus table.
ManagedStatusUpdate Interval	NOT NULL	Integer	Interval in seconds at which ncp_model scans the NCIM managedStatus table for changes. This is the maximum time the poller should take to react to changes in managed status made in any of the following GUIs: Network Views, Network Hop View, Structure Browser. Default value 30 seconds.

Any combination of the flags `ChassisCreationEvents`, `IpInterfaceCreationEvents`, and `MaintenanceStateEvents` can be turned on and off. The default is for all three to be disabled.

Note: If you have a network that contains routers with a large number of IP addresses, then enabling the `IpInterfaceCreationEvents` flag can might generate a large number of events in the Object Server.

model.profilingData

The model.profilingData table stores data associated with time and memory expended during the discovery. This table includes information on how long it took to transfer the discovery profiling data to the NCIM topology database.

Table 189. model.profilingData database table schema			
Column name	**Constraints**	**Data type**	**Description**
`BatchStartTime`	• PRIMARY KEY • NOT NULL • UNIQUE	Integer	The time that a batch update from the Discovery engine, ncp_disco, started.
`BatchStartSize`	NOT NULL	Integer	Number of records in the batch received.
`BatchStartMem`	NOT NULL	64-bit integer	Memory usage when batch started.

Table 189. model.profilingData database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
BatchEndTime		Integer	The time a batch update from the Discovery engine, ncp_disco, ended.
BatchEndSize		Integer	Number of records at the end. **Note:** This value could be larger than at the start if subsequent batches got merged in.
BatchEndMem		64-bit integer	Memory usage when batch ended.
EntityCount		Integer	Number of entities after the batch update.
ChassisCount		Integer	Number of chassis devices after the batch update.
InterfaceCount		Integer	Number of interfaces after the batch update.

model.statistics table

The model.statistics table stores information about previous discoveries.

Table 190. model.statistics database table schema			
Column name	**Constraints**	**Data type**	**Description**
TopologyCount	• PRIMARY KEY • NOT NULL • UNIQUE	Long	A count of the number of times the topology has been sent from DISCO to MODEL.
TopologySendFinished		Integer	Indicates whether DISCO has finished transferring the topology to MODEL. This column is set to 0 when the SendTopologyToModel.stch stitcher begins sending the topology, and set to 1 when it has completed sending the topology.
InsertCount		Long	The number of entities inserted into the topology.
UpdateCount		Long	The number of entities updated in the topology.
DeleteCount		Long	The number of entities deleted from the topology.

Failover database

Failover recovery with the failover database is not to be confused with agent and finder failover recovery, which are configured directly from the disco.config table. When selected, agent and finder failover recovery operate regardless of whether recovery with the failover database is implemented.

If the m_WriteTablesToCache column of the disco.config table is set to 1 (true), data is cached during the discovery process to enable data recovery in the event that the Discovery engine, ncp_disco, fails. A discovery running in this mode is slower than a standard discovery, because of the extra time required to store data on the disk throughout the discovery process.

Ignored cached data

If DISCO is restarted in failover recovery mode, any cached data for a group of tables are ignored.

The cached data for the following tables are ignored when DISCO is restarted in failover recovery mode:

- disco.config
- disco.managedProcesses
- disco.agents
- The entire scope database
- failover.config
- failover.doNotCache
- failover.restartPhaseAction

For the above tables, only the insertions specified in the schema file at the time of the restart are registered.

The failover database schema

The failover database is defined in $NCHOME/etc/precision/DiscoSchema.cfg. Its fully qualified database table names are: failover.config; failover.status; failover.findRateDetails; failover.doNotCache; failover.restartPhaseAction.

failover.config table

There must never be more than one insert into the failover.config table.

Table 191. failover.config database table schema

Column name	Constraints	Data type	Description
`m_InitialiseFromCache`	Externally defined Boolean data type	Boolean integer	Flag indicating whether to use the data that already exists in the cache: • 0: Do not use cached data • 1: Use cached data if any exists
`m_NumberOfRetries`		Integer	The number of times to try using the cached data before giving up (that is, the number of subsequent times that DISCO can be restarted before starting with a clean slate). If no value is specified, DISCO always starts with clear databases.

Table 191. failover.config database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`m_StoreEveryNthDevice`	Default = 10	Integer	How often the findRateDetails table is to be updated. After the specified number of devices have been found the table is updated.

failover.status table

The failover.status table displays the number of times that the DISCO process has attempted to restart with cached data. This table is active, so you must not configure inserts into it.

Table 192. failover.status database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_NumberOfAttempts`	• NOT NULL • PRIMARY KEY	Integer	The number of times that the DISCO process has attempted to restart with cached data. This column is set to 1 when DISCO is first run in failover recovery mode and incremented each time DISCO is subsequently run in failover mode.

failover.findRateDetails table

The findRateDetails table gives details of devices that have been found at a certain point in the discovery. This table is active and inserts must not be made in the schema file; the table is populated automatically.

Table 193. failover.findRateDetails database table schema			
Column name	**Constraints**	**Data type**	**Description**
`m_StartTime`	• NOT NULL • PRIMARY KEY	Text	The time at which the first device was found.
`m_LastFindTime`		Text	The time at which the last device was found.
`m_DevicesFound`		Integer	The number of devices found so far.

failover.doNotCache table

To prevent caching a given table, you can specify its name in the doNotCache table. This ensures that unnecessary cache files are not created, such as those for temporary tables defined within stitchers.

Table 194. failover.doNotCache database table schema

Column name	Constraints	Data type	Description
`m_DatabaseName`	NOT NULL	Text	The name of any database that is not to be cached during failover recovery. The following tables must be cached in order to use the failover recovery mode, and therefore must not be listed in this table: • disco.status • failover.status The following tables must be cached, and therefore must not be listed in this table: • The agent despatch and returns tables. • finders.processing • translations.ipToBaseName
`m_TableName`	NOT NULL	Text	The name of the table within the database specified in m_DatabaseName that is not to be cached. Use * to indicate all the tables of the database.

failover.restartPhaseAction table

The restartPhaseAction table contains the set of stitchers that are executed when restarting in a given discovery phase. Multiple stitchers can be specified, but they are executed in an arbitrary order. It is recommended that at least the FinalPhase stitcher is executed when restarting in the topology creation phase.

Table 195. failover.restartPhaseAction database table schema

Column name	Constraints	Data type	Description
`m_RestartPhase`	NOT NULL	Integer	The phase in which DISCO is restarted.
`m_ExecuteStitcher`	NOT NULL	Text	The stitcher that is to be executed in this phase.

Example failover database configuration

This example uses OQL commands to insert configuration values into the failover database tables that are appended to the DiscoConfig.cfg file to configure DISCO when it is launched.

Example configuration of the failover.config table

This example uses OQL commands to insert configuration values into the failover.config table.

For this configuration of the failover.config table, data already in the cache is used. The Discovery engine, ncp_disco, can be restarted up to three times before cached data is ignored. These values are used only when disco.config.m_WriteTablesToCache=1.

```
insert into failover.config
(
        m_InitialiseFromCache,
        m_NumberOfRetries
)
values
(  1, 3  );
```

Example configuration of the failover.doNotCache table

This example uses OQL commands to insert configuration values into the failover.doNotCache table. The disco.config table and all tables of the instrumentation database are not cached.

```
insert into failover.doNotCache
(
        m_DatabaseName,
        m_TableName
)
values
(
        'disco', 'config'
);

insert into failover.doNotCache
(
        m_DatabaseName, m_TableName
)
values
(
        'instrumentation', '*'
);
```

Agent Template database

The databases of each discovery agent are based on a template called the agentTemplate database.

The agentTemplate database is defined in $NCHOME/etc/precision/DiscoSchema.cfg, and its fully qualified database table names are: agentTemplate.despatch and agentTemplate.returns.

Discovery agent despatch table

When a device has been interrogated by the Details agent, it is passed to the Associated Address agent to check whether it has already been discovered. If the device has not been discovered, the device details are processed and sent by a stitcher to the despatch table of the appropriate agent.

The despatch table is described in Table 196 on page 413.

When the device details are placed in the despatch table, the agent attempts to retrieve connectivity information pertaining to the device.

Table 196. agentTemplate.despatch database table schema

Column name	Constraints	Data type	Description
m_Name	PRIMARY KEY NOT NULL	Text	Unique name of an entity on the network.

Table 196. agentTemplate.despatch database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_UniqueAddress	NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
m_ManagerId	PRIMARY KEY NOT NULL	Text	Manager of the device. If the device is accessed directly, this is set to " ". By default, this is set to " ".
m_Protocol		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
m_ObjectId		Text	Textual representation of the device class (an ASN.1 address).
m_SnmpAccessIP		Text	If present, overrides the IP address used for SNMP access to devices using the Helper Server.
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.
m_HaveAccess	Externally defined Boolean data type	Boolean Integer	Flag indicating whether there is SNMP access to the device: • (1) Have Access • (0) No Access

Discovery agent returns table

Returned device connectivity details are placed in the returns table of the agent. These details are used to populate the topology databases.

The returns table is described in Table 197 on page 414.

Table 197. agentTemplate.returns database table schema			
Column name	**Constraints**	**Data type**	**Description**
m_Name	NOT NULL	Text	Unique name of an entity on the network.

Table 197. agentTemplate.returns database table schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_ManagerId	PRIMARY KEY NOT NULL	Text	Manager of the device. If the device is accessed directly, this is set to " ". By default, this is set to " ".
m_UniqueAddress	NOT NULL	Text	A string that uniquely identifies this entity. The content of this field is unconstrained, and might be an IP address or an element management system (EMS) key.
m_Protocol		Integer	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
m_ObjectId		Text	Textual representation of the device class (an ASN.1 address).
m_HaveAccess	Externally defined Boolean data type	Boolean Integer	Flag indicating whether there is SNMP access to the device: • (1) Have Access • (0) No Access
m_ExtraInfo	Externally defined vblist data type	Object	Any extra information specified by the user in the agent definition file.
m_LocalNbr	Externally defined neighbor data type	Object	Direct neighbors (interfaces).
m_RemoteNbr	Externally defined nbrsNeighbor data type	Object	Remote neighbors connected to interfaces.
m_UpdAgent		Text	The agent that updated this device.
m_SnmpAccessIP		Text	If present, overrides the IP address used for SNMP access to devices using the Helper Server.
m_AddressSpace		Text	The name of the NAT address space to which the device belongs. This value is set in the translations.NATAddressSpaceIds table. If the discovery is not using NAT, or if the device is in the public domain, this value is NULL.

Table 197. agentTemplate.returns database table schema (continued)

Column name	Constraints	Data type	Description
m_LastRecord	Externally defined Boolean data type	Boolean integer	Is this the last record for this entity: • (1) True • (0) False

Chapter 13. Polling databases

Use this information to understand the structure of databases used for polling.

NCMONITOR databases

The NCMONITOR schema hosts a number of databases used by polling.

SNMP tables for polling in the ncmonitor database

The SNMP tables in the ncmonitor database are used by the polling engine, ncp_poller, to store information on how to access each discovered device using SNMP.

Both the ncp_dh_snmp and ncp_poller processes use the ncmonitor database. However, only the ncp_dh_snmp process populates the database; the ncp_poller process treats it as read-only. Hence, you must have discovered a device using SNMP to monitor it using SNMP.

The ncmonitor database is defined in `$NCHOME/etc/precision/DbLogins.`*`DOMAIN`*`.cfg`, where *`DOMAIN`* is the domain that contains the discovered devices.

The ncmonitor database has the following tables:

- ncmonitor.snmpTarget
- ncmonitor.snmpAccess
- ncmonitor.snmpv1Sec
- ncmonitor.snmpv3Sec
- ncmonitor.snmpUser

ncmonitor.snmpTarget table

The snmpTarget table lists each IP address that Network Manager recognizes.

Table 198. ncmonitor.snmpTarget database table

Column name	Constraints	Data type	Description
targetid	• PRIMARY KEY • NOT NULL	Text	Unique identifier for the target.
netaddr		Text	IP address of the target.
readaccessid	FOREIGN KEY	Text	Refers to the snmpaccess table. Provides access details used to perform SNMP Get and GetNext operations for this target.
writeaccessid	FOREIGN KEY	Text	Refers to the snmpaccess table. Provides access details used to perform SNMP Set operations for this target.
snmpgetbulk		Boolean integer	Flag indicating whether GetNext operations will be attempted when appropriate; for example, when using SNMPv2 or SNMPv3, and performing a table walk.
snmpthrottleid		Text	Throttling details used to control the rate at which requests will be made to this target when performing table walk operations.

Table 198. ncmonitor.snmpTarget database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
createtime		Text	Timestamp recording the time this target was created.
lastupdate		Text	Timestamp recording the time any detail for this target was last modified.
domain		Text	Domain to which this target belongs.

ncmonitor.snmpAccess table

The snmpAccess table provides details of SNMP access.

Table 199. ncmonitor.snmpAccess database table			
Column name	**Constraints**	**Data type**	**Description**
accessid	• PRIMARY KEY • NOT NULL	Text	Unique identifier for these SNMP access details.
version		Enumerated value	SNMP version to be used. Possible value are: • 0: SNMPv1 • 1: SNMPv2 • 3: SNMPv3
remoteport		Integer	UDP port to which SNMP packets will be sent.
retries		Integer	Number of retries before giving up.
timeout		Integer	Number of milliseconds before retrying an SNMP request .
accesslevel		Enumerated value	Flag indicating level of access provided. Possible values are: • 1: read • 2: write

ncmonitor.snmpv1Sec table

The snmpv1Sec table is populated only for rows in the snmpAccess table that relate to SNMPv1 and SNMPv2.

Table 200. ncmonitor.snmpv1Sec database table			
Column name	**Constraints**	**Data type**	**Description**
accessid	FOREIGN KEY	Text	Refers to the details for which SNMPv1 or SNMPv2-specific detail is being provided.
community		Text	Community string to use when sending requests using these details.

Table 200. ncmonitor.snmpv1Sec database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
encrypted		Boolean integer	Flag indicating whether the community string is encrypted. Possible values are: • 0: not encrypted • 1: encrypted

ncmonitor.snmpv3Sec table

The snmpv3Sec table is populated only for rows in the snmpAccess table that relate to SNMPv3.

Table 201. ncmonitor.snmpv3Sec database table			
Column name	**Constraints**	**Data type**	**Description**
accessid	FOREIGN KEY	Text	Refers to the details for which SNMPv3-specific detail is being provided.
userid	FOREIGN KEY	Text	Refers to the userid field in the snmpusmuser table. This is the user to use when sending SNMP requests using these details.
securitylevel		Enumerated value	Flag indicating SNMPv3 security level. Possible values are: • noAuthNoPriv • authNoPriv • authPriv
defaultcontext		Text	SNMPv3 contextName to be used when not explicitly specified by a discovery agent.

ncmonitor.snmpUser table

The snmpUser table provides a list of SNMP user details to be used by the SNMPv3 protocol.

Table 202. ncmonitor.snmpUser database table			
Column name	**Constraints**	**Data type**	**Description**
userid	• PRIMARY KEY • NOT NULL	Text	Unique identifier for this user.
username		Text	USM username.
authpass		Text	Authentication password.
privpass		Text	Privacy password. This is used for encrypting the SNMPv3 payload.
authtype		Text	Encryption method to be used for the SNMPv3 authentication header.
privtype		Text	Encryption method to be used for the payload.

Table 202. ncmonitor.snmpUser database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
encrypted		Boolean integer	Flag indicating authpass and authtype fields are encrypted. Possible values are: • 0: not encrypted • 1: encrypted

Ping polling status tables

The NCMONITOR ping polling status tables enable diagnostic operations to be performed on network ping polling.

expectedIps table

The expectedIps table contains a list of IP addresses expected to be discovered by Network Manager for a particular domain. It is populated using the ncp_upload_expected_ips.pl script.

The following table lists the columns in the expectedIps table.

Table 203. ncmonitor.expectedIps database table	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.

pollLog table

The pollLog table stores the latest snapshot of the status of the Polling engine. It is populated using the ncp_ping_poller_snapshot.pl script which queries ncp_poller, and transfers the results to this table.

Each row in this table corresponds to a single entity that is within the defined scope of a single active polling policy. The fields in the table can be divided into three conceptual groupings:

"Entity information" on page 420
This entity information can be used to cross-reference with other NCIM topology database tables.

"Managed status information" on page 421
This is the managed status being applied by the poll policy.

"Latest poll state information" on page 422
This is the latest poll state for the current entity and policy.

Entity information

Fields in the pollLog table that store entity information are described below.

Table 204. ncmonitor.pollLog database table entity information fields	
Column name	**Description**
entityId	ID of the entity to ping, as defined in the NCIM topology database entityData table.
policyId	ID of the relevant ping policy, as defined in the NCMONITOR policy table.

Table 204. ncmonitor.pollLog database table entity information fields (continued)	
Column name	**Description**
mainNodeEntityId	ID of the main node that the entity belongs to, as defined in the NCIM topology database entityData table. For Chassis Ping polls, this is the same as the entityId. For Interface Ping polls, this is the ID of the main node containing the interface.
entityType	As defined in the NCIM topology database entityType table, this field can take one of the following values: • 1: Chassis Ping polls • 2 Interface Ping polls
ip	IP address to which the ICMP ping packet was sent. This is the accessIPAddress of the interface or chassis entity identified by entityId.
mainNodeAddress	IP address of the main node that the entity belongs to, as found in the NCIM topology database entityData table. This is the accessIPAddress of the chassis entity identified by mainNodeEntityId.
ifIndex	The ifIndex of the relevant interface might be available for Interface Ping polls. This can be NULL for any ping poll.
domainMgrId	ID of the relevant domain, as found in the NCIM topology database domainMgr table.

Managed status information

Fields in the pollLog table that store managed status information are described below.

Table 205. ncmonitor.pollLog database table managed status information fields	
Column name	**Description**
isManaged	Indicates whether the entity is being polled by this policy? • 0: False • 1: True
entityStatus	Managed status of the entity identified by entityId that is being used by the poller. The value of this field is set to 0 if managed, whether explicitly listed in the NCIM managedStatus table or not. **Note:** This is the status at the time of the snapshot, and therefore can differ from the contents of the NCIM managedStatus table if it is dynamically altered.
mainNodeStatus	Managed status of the entity identified by mainNodeEntityId that is being used by the poller. Interfaces within an unmanaged main node are also unmanaged. The value of this field is set to 0 if managed, whether explicitly listed in the NCIM managedStatus table or not. **Note:** This is the status at the time of the snapshot, and therefore can differ from the contents of the NCIM managedStatus table if it is dynamically altered.
entityChangeTime	Timestamp of the last change to the entityStatus field. Defaults to a zero timestamp if unused.

Table 205. ncmonitor.pollLog database table managed status information fields (continued)	
Column name	**Description**
mainNodeChangeTime	Timestamp of the last change to the mainNodeStatus field. Defaults to a zero timestamp if unused.

Latest poll state information

Fields in the pollLog table that store latest poll state information are described below.

Table 206. ncmonitor.pollLog database table latest poll state information fields	
Column name	**Description**
lastPollFailure	Last time at which a ping poll failure event was raised. Defaults to a zero timestamp if no poll failures have been raised since the poller was started.
lastPollInterval	Duration of the last complete polling cycle, in seconds. This field is NULL if the policy is not actively monitoring the entity or a complete poll cycle has not finished since the poller started. **Note:** The system must be on the third polling interval when the snapshot is taken.
timeSinceLastPoll	Number of seconds since the last poll, at the time the snapshot was taken.
snapshotTime	Time at which the data was retrieved from the poller. This is a zero timestamp if the policy is not actively monitoring the entity.

pollLogSummary table

This table stores a summary of the results for each snapshot written to the pollLog table for a domain, generated using the views listed in the following sections. It is populated using the ncp_ping_poller_snapshot.pl script which queries the poller, and transfers the results to this table.

The following table lists the columns in the pollLogSummary table.

Note: The ncp_ping_poller_snapshot.pl script never clears out existing data from this table, so the table can grow. If required, data that is no longer of interest can be removed by filtering against the domainMgrId or the summaryTimestamp fields.

Table 207. ncmonitor.pollLogSummary database table	
Column name	**Description**
domainMgrId	ID of the relevant domain, as found in the NCIM domainMgr table.
domainName	Name of the domain identified by the domainMgrId.
undiscoveredIps	Count of IP addresses returned from the undiscoveredIps view for this domain after the snapshot was loaded to the pollLog.
unmonitoredIps	Count of IP addresses returned from the unmonitoredIps view for this domain after the snapshot was loaded to the pollLog.
unmanagedIps	Count of IP addresses returned from the unmanagedIps view for this domain after the snapshot was loaded to the pollLog.
unpolledFor15MinutesIps	Count of IP addresses returned from the unpolledFor15MinutesIps view for this domain after the snapshot was loaded to the pollLog.

Table 207. ncmonitor.pollLogSummary database table (continued)	
Column name	**Description**
delayedPollPolicies	Count of IP addresses returned from the delayedPollPolicies view for this domain after the snapshot was loaded to the pollLog table.
summaryTimestamp	Timestamp indicating when the summary was generated.

undiscoveredIps view

The undiscoveredIps view lists any IP addresses that were are not discovered by Network Manager and therefore are not listed in the NCIM topology database, but that you expected to discover. The IP addresses listed in this table are those that were loaded into the expectedIps table but are not present in NCIM.

The following table lists the columns in the undiscoveredIps view.

Table 208. ncmonitor undiscoveredIps view	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column..

unmonitoredIps view

The unmonitoredIps view uses the latest poller snapshot from the pollLog table to list any IP addresses from the expectedIps table that are not currently being polled because they are not in the scope of any active ping policy.

The following table lists the columns in the unmonitoredIps view.

Table 209. ncmonitor unmonitoredIps view	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

unmanagedIps view

The unmanagedIps view uses the latest poller snapshot from the pollLog table to list any IP addresses from the expectedIps table that are in the scope of active ping policies, but that are not being monitored because they are unmanaged. This is based on the managed status known to the poller at the time of the snapshot.

The following table lists the columns in the unmanagedIps view.

Table 210. ncmonitor unmanagedIps view

Column name	Description
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

unpolledFor15MinutesIps view

The unpolledFor15MinutesIps view uses the latest poller snapshot from the pollLog table to list any IP addresses from the expectedIps table that have not been ping polled at all in the last 15 minutes. This includes any IP addresses that are unmanaged or outside the scope of the configured ping polling policies.

The following table lists the columns in the unpolledFor15MinutesIps view.

Table 211. ncmonitor unpolledFor15MinutesIps view

Column name	Description
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

delayedPollPolicies view

The delayedPollPolicies view uses the latest poller snapshot from the pollLog table to list all active ping policies are lagging behind schedule.

If the current or previous time interval in the poll cycle (measured from the time at which the poller snapshot was taken) are greater than twice the configured poll interval, the entity and the relevant policy will be listed in this view. This excludes entities that are out of poll scope (see unmonitoredIps view) or that have been unmanaged (see unmanagedIps view), and only applies to the latest snapshot in the pollLog table.

The following table lists the columns in the delayedPollPolicies view.

Table 212. ncmonitor delayedPollPolicies view

Column name	Description
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
entityType	As defined in the NCIM entityType table, this will be: • 1 for Chassis Ping polls • 2 for Interface Ping polls
policy	The policyName of the ping policy, as found in the NCMONITOR policy table.
configuredPollInterval	The configured pollInterval of the ping policy, as found in the NCMONITOR policy table.
lastPollInterval	The duration of the last complete polling cycle, in seconds.
timeSinceLastPoll	The number of seconds since the last poll, at the time the snapshot was taken.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

discoveredIps view

The discoveredIps view lists all IP addresses in the NCIM topology database, together with details of the associated device.

The following table lists the columns in the discoveredIps view.

Table 213. ncmonitor discoveredIps view

Column name	Description
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.

Table 213. ncmonitor discoveredIps view (continued)	
Column name	**Description**
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
mainNodeMgdStatus	The current managed status of the main node, as found in the NCIM managedStatus table.
entityMgdStatus	The current managed status of the entity, as found in the NCIM managedStatus table.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

managementInterfaceIps view

The managementInterfaceIps view lists SNMP management interface IP addresses for all devices for which Network Manager obtained SNMP access. It does not list the IP addresses of chassis for which no SNMP access was obtained.

For SNMP-accessible devices, the IP address assigned to the chassis by Network Manager must also be the IP address of an interface on that device. This view therefore displays the set of IP addresses which can be monitored by both chassis ping polls and interface ping polls.

The following table describes the columns in the managementInterfaceIps view.

Table 214. ncmonitor managementInterfaceIps view	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
entityId	The entityId of the interface, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
ifIndex	The ifIndex of the interface from the NCIM interface table.
mainNodeMgdStatus	Managed status of the chassis entity identified by the mainNodeEntityId column from the last poller snapshot.
interfaceMgdStatus	Managed status of the interface entity identified by the mainNodeEntityId column from the last poller snapshot.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.

Table 214. ncmonitor managementInterfaceIps view (continued)	
Column name	**Description**
domainName	The name of the domain identified by the domainMgrId column.

chassisOnlyIps view

The chassisOnlyIps view lists the IP addresses which can only be monitored with the chassis ping polls, as no interfaces with IP addresses have been discovered on these devices. This is usually the case when Network Manager failed to obtain SNMP access to the device, although it can also depend on the discovery configuration.

The following table lists the columns in the chassisOnlyIps view.

Table 215. ncmonitor chassisOnlyIps view	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.
mainNodeMgdStatus	Managed status of the chassis entity identified by the mainNodeEntityId column from the last poller snapshot.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

unpollableIps view

The unpollableIps view lists the IP addresses that the poller will not attempt to ping poll. These IP addresses can be monitored using SNMP poll policies.

These are only the secondary IP addresses of multinet interfaces, where a single interface has multiple IP addresses. Ping polls work from the accessIPAddress field of the NCIM chassis and interface tables, and thus only a single IP address per interface can be monitored using ping polls.

The following table lists the columns in the unpollableIps view.

Table 216. ncmonitor unpollableIps view	
Column name	**Description**
ip	A dot-notation IPv4 address that is expected to have been discovered and added to the NCIM topology database.
mainNode	The entityName of the main node, as found in the NCIM entityData table.
mainNodeEntityId	The entityId of the main node, as found in the NCIM entityData table.
class	The entityClass of the main node, as defined by the NCIM classMembers and entityClass tables.

Table 216. ncmonitor unpollableIps view (continued)	
Column name	**Description**
ifIndex	ifIndex The ifIndex of the interface from the NCIM interface table.
interfaceAccessIp	interfaceAccessIp The primary, pollable IP address for the multi-net interface that has this ip as a secondary IP address.
mainNodeMgdStatus	Managed status of the chassis entity identified by the mainNodeEntityId column from the last poller snapshot.
interfaceMgdStatus	interfaceMgdStatus Managed status of the interface entity identified by mainNodeEntityId from the last poller snapshot.
domainMgrId	The ID of the relevant domain, as found in the NCIM topology database domainMgr table.
domainName	The name of the domain identified by the domainMgrId column.

NCPOLLDATA database

The NCPOLLDATA database stores raw and historical polled data. It is used by dashboard widgets and reports to present polling data.

The NCPOLLDATA database

The NCPOLLDATA database stores a data that is used by the Polling engine to administer polling and to administer storage and pruning of historical poll data. It also stores raw collected by the pollers and historical poll data that is derived from the raw data.

The NCPOLLDATA database tables fall into the following categories:

- Tables indicating which entities to poll.
- Tables indicating how to poll.
- Tables containing raw and historical poll data.
- Tables containing information on how to prune raw and historical poll data.
- Tables that are used by the Apache Storm real-time computation system when it is managing the aggregation of raw poll data into historical poll data.
- Views that are used by the **Top Performers** GUI to support the visualization of raw and historical poll data.

Tables indicating which entities to poll

The following tables contain subsets of data from the NCIM topology database. The Polling engine, ncp_poller, uses data from these tables to determine which entities to poll.

- **domainMgr**
- **monitoredEntity**
- **monitoredChassis**
- **monitoredInterface**

Tables indicating how to poll

The following tables contain subsets of data from the NCMONITOR database. The Polling engine, ncp_poller, uses data from these tables to determine how to poll.

- **poller**
- **template**

- **policy**
- **poll**
- **monitoredInstance**
- **pollInstance**
- **monitoredObject**

Tables containing raw and historical poll data

The following tables store raw and historical poll data.

- **pollData**
- **pdEwmaForDay**
- **pdEwmaForWeek**
- **pdEwmaForMonth**
- **pdEwmaForYear**

Note: The **pollBatch** table is related to the **pollData** table and contains identifiers used in the **pollData** table records.

Tables containing partition information

The following tables contain data on partitioning. The NCPOLLDATA tables that contain raw and historical poll data are partitioned to enable data pruning. These partitioning tables are used by the Polling engine, ncp_poller to manage the partitioning of raw and historical poll data tables.

- **pollDataPartitions**
- **detachedPartitions**
- **partitionSizes**
- **partitionLog**

Tables used by Apache Storm

The following tables are used by the Apache Storm real-time computation system to manage the aggregation of raw poll data into historical poll data.

- **getPollDataLog**
- **master**
- **mastershipAuditTrail**
- **mastershipCheck**

Views used by the GUI

The following views are used by the **Top Performers** GUI to support the visualization of raw and historical poll data.

- KNP_POLL_DATA_COLLECTION
- pollItemView
- PERFORMANCE_DATA
- PERFORMANCE_DATA_DAILY
- PERFORMANCE_DATA_WEEKLY
- PERFORMANCE_DATA_MONTHLY
- PERFORMANCE_DATA_YEARLY
- CHASSIS_PERFORMANCE_DATA
- NETWORK_VIEW_PERF_DATA
- NETWORK_VIEW_PERF_DATA_DAILY

- NETWORK_VIEW_PERF_DATA_WEEKLY
- NETWORK_VIEW_PERF_DATA_MONTHLY
- NETWORK_VIEW_PERF_DATA_YEARLY
- POLICIES_PER_NETWORK_VIEW

NCPOLLDATA queries

Use these sample NCPOLLDATA queries to investigate issues associated with partioning of poll data tables.

Logging in to the NCPOLLDATA database

Log in to the NCPOLLDATA database to run an SQL query that retrieves polling data.

To log in to NCPOLLDATA using ncp_oql you must log in using the ncim service and specify the NCPOLLDATA database identifier. You must also provide a valid NCIM user name and password. The default user name for the NCIM database user is ncim. The default password is ncim.

To log in to NCPOLLDATA enter the following command:

```
ncp_oql -domain DOMAIN -service ncim -username USERNAME -password PASSWORD
```

Where:

- *DOMAIN* is any Network Manager domain. The pollers run across all domains so it does not matter which domain you choose.
- *USERNAME* is the user name for the NCIM database user. The default is ncim.
- *PASSWORD* is the password for the NCIM database user. The default is ncim.

Show partitions allocated to a specific raw or historical poll data table

This query lists the partitions allocated to a specific raw or historical poll data table. This query is useful if you want to investigate whether all partitions are being created properly.

The sample query provided shows partitions allocated to the raw poll data , pollData. By default, this table stores approximately an hour of raw poll data, and the table is partitioned into 8 partitions of 10 minutes each. Check the results of this query to make sure that the upper and lower ranges of the partition reflect expected values according to the clock on the server hosting the database. If these times are not current then either the relevant instance of the Polling engine, ncp_poller, is not runnning, or there is a problem creating new partitions.

Example

This example query displays sample SQL to show partitions allocated to the raw poll data , pollData.

```
select TABNAME,DATAPARTITIONNAME,LOWTIME,HIGHTIME
from ncpolldata.pollDataPartitions
WHERE TABNAME = 'POLLDATA'
ORDER BY HIGHTIME
```

The table below describes this query.

Table 217. Description of the query	
Line numbers	**Description**
1	Specify the data to show in the results. • TABNAME: name of thje table to which the partitions belong. • DATAPARTITIONNAME: name of the partitions. • LOWTIME: lower time limit for the partition. • HIGHTIME: upper time limit for the partition.
2	Specify the pollDataPartitions table as the driving table for this query.
3	Limit the partition data retrieved to those partitions related to the pollData table.
4	List results in order of the HIGHTIME column. This orders the data by latest partition first.

Results

The table below shows sample results for this query.

Table 218. Results of the query			
Table name	**Partition name**	**Low time**	**High time**
POLLDATA	PART_1439376000	2015-08-13 13:00:00	2015-08-13 13:10:00
POLLDATA	PART_1439376600	2015-08-13 13:10:00	2015-08-13 13:20:00
POLLDATA	PART_1439377200	2015-08-13 12:00:00	2015-08-13 12:10:00
POLLDATA	PART_1439377800	2015-08-13 12:10:00	2015-08-13 12:20:00
POLLDATA	PART_1439378400	2015-08-13 12:20:00	2015-08-13 12:30:00
POLLDATA	PART_1439374200	2015-08-13 12:30:00	2015-08-13 12:40:00
POLLDATA	PART_1439374800	2015-08-13 12:40:00	2015-08-13 12:50:00
POLLDATA	PART_1439374800	2015-08-13 12:50:00	2015-08-13 13:00:00

Show which partitions have recently been detached and dropped

This query shows the recent history of which partitions have been detached and then dropped and also those which have not yet been detached but are pending a drop soon. This query is useful if you want to investigate whether partitions are being detached and dropped correctly.

When running this query, check the value of the `status` field. It should normally cycle through the following states: it begins as `pending` then becomes `detaching` and ends up as `dropped`. If the `status` value remains as `pending` for an extended length of time then that would indicate that the poller pruning thread is not running. If it remains at `detaching` for an an extended length of time then this is an indication that there are problems detaching partitions.

The sample query provided shows data related to partitions being detached from the raw poll data table, `pollData`. Check the `status` value for these partitions; any `status` value that does not cycle to `dropped` but stays at `pending` or `detaching` might indicate an issue that requires further investigation.

Example

This example query displays data related partitions being detached from the raw poll data , `pollData`.

```
select TABNAME,DETACHTIME,CLIENTID,DATAPARTITIONNAME,LOWTIME,HIGHTIME,STATUS
from ncpolldata.detachedPartitions
WHERE TABNAME = 'POLLDATA'
ORDER BY HIGHTIME
```

The table below describes this query.

Table 219. Description of the query

Line numbers	Description
1	Specify the data to show in the results. • TABNAME: table name to which the partitions belong. • DETACHTIME: time at which the partition detach operation was requested . • CLIENTID: name of the poller that handled the detach operation. • DATAPARTITIONNAME: name of the related partition. • LOWTIME: lower time limit of data in that partition. • HIGHTIME: upper time limit of data in that partition. • STATUS: takes one of the following values: `pending`, `detaching` or `dropped`.
2	Specify the detachedPartitions table as the driving table for this query.
3	Limit the partition data retrieved to those partitions detached from the pollData table.
4	List results in order of the HIGHTIME column. This orders the data by latest partition first.

Results

The table below shows sample results for this query.

Table name	Detach time	Client ID	Partition name	Low time	High time	Status
POLL DATA	2015-08-13 11:40:14	ncp_poller_ default_POLLDATA	PART_143 9458800	2015-08-13 10:30:00	2015-08-13 10:40:00	dropped
POLL DATA	2015-08-13 11:50:20	ncp_poller_ default_POLLDATA	PART_143 9459400	2015-08-13 10:40:00	2015-08-13 10:50:00	dropped
POLL DATA	2015-08-13 12:00:15	ncp_poller_ default_POLLDATA	PART_143 9460000	2015-08-13 10:50:00	2015-08-13 11:00:00	dropped
POLL DATA	2015-08-13 12:10:20	ncp_poller_ default_POLLDATA	PART_143 9460600	2015-08-13 11:00:00	2015-08-13 11:10:00	dropped
POLL DATA	2015-08-13 12:20:15	ncp_poller_ default_POLLDATA	PART_143 9461200	2015-08-13 11:10:00	2015-08-13 11:20:00	dropped
POLL DATA	2015-08-13 12:30:10	ncp_poller_ default_POLLDATA	PART_143 9461800	2015-08-13 11:20:00	2015-08-13 11:30:00	dropped

Table name	Detach time	Client ID	Partition name	Low time	High time	Status
POLL DATA	2015-08-13 12:40:15	ncp_poller_ default_POLLDATA	PART_143 9462400	2015-08-13 11:30:00	2015-08-13 11:40:00	dropped
POLL DATA	2015-08-13 12:50:10	ncp_poller_ default_POLLDATA	PART_143 9463000	2015-08-13 11:40:00	2015-08-13 11:50:00	dropped
POLL DATA	2015-08-13 13:00:15	ncp_poller_ default_POLLDATA	PART_143 9463600	2015-08-13 11:50:00	2015-08-13 12:00:00	dropped
POLL DATA	2015-08-13 13:10:11	ncp_poller_ default_POLLDATA	PART_143 9464120	2015-08-13 12:00:00	2015-08-13 12:10:00	dropped

Show log messages for recently attached and detached partitions

This query displays log messages for recently attached and detached partitions.

You are unlikely to run this query unless prompted to by log errors or alerts. However, if you run this query and you notice that the `logmsg` field has the value %SQLCODE% this could mean that the database is having problems partitioning.

The sample query provided shows log messages for partitions related to the raw poll data, pollData.

Example

This example query displays log messages for partitions related to the raw poll data, pollData.

```
select LOGID,TABNAME,CLIENTID,LOGTIME,LOGMSG
from ncpolldata.partitionLog
WHERE TABNAME = 'POLLDATA'
ORDER BY LOGTIME, LOGID
```

The table below describes this query.

Table 220. Description of the query

Line numbers	Description
1	Specify the data to show in the results. • LOGID: identifier for this log message. • TABNAME: table name to which the partitions belong. • CLIENTID: name of the poller that handled the detach operation. • LOGTIME: time the message was logged. • LOGMSG: content of the log message.
2	Specify the partitionLog table as the driving table for this query.
3	Limit the partition data retrieved to those partitions related to the pollData table.
4	List results in order of the time the message was logged, and then by the content of the log message.

Results

The table below shows a subset of results for this query.Results of the query

Log ID	Table name	Client ID	Log time	Log message
10101	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:14	Start ATTACH_POLLDATA_PARTITION for: POLLDATA with pollTime: 1439372404
10102	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:14	Checking for partition in: POLLDATA with highValue: 1439373600
10103	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:14	Created partition in: POLLDATA with highValue: 1439373600
10104	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:14	Partition pending detach: PART_1439368800 id 6 with highValue: 1439368800
10105	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:14	End ATTACH_POLLDATA_PARTITION for: POLLDATA with pollTime: 1439372404
10106	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:19	Start DETACH_POLLDATA_PARTITION
10107	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:19	Detaching partition: PART_1439368800 with hightValue: 1439368800 into: POLLDATA_PART_1439368800
10108	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:19	Finished waiting for detach of: PART_1439368800 from: POLLDATA
10109	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:40:19	Dropped detached table: POLLDATA_PART_1439368800
101010	POLLDATA	ncp_poller_default_POLLDATA	2015-08-12 10:50:09	End DETACH_POLLDATA_PARTITION

OQL databases

The embedded OQL polling databases provide a number of polling configuration options.

config database for polling

The config database is used by the polling engine, ncp_poller, for a variety of purposes, including diagnostic and debugging purposes, facilitating failover in high availability deployments, debugging the MIB grapher, and configuring the storage limit for historical performance data.

The config database for polling is defined in `$NCHOME/etc/precision/NcPollerSchema.cfg`.

The config database has the following tables:

- config.properties
- config.failover
- config.realTimeControl
- config.pruning

config.properties table

The config.properties table provides the option to configure a number of polling settings.

You can set the values in the config.properties table by editing the following file: `$NCHOME/etc/precision/NcPollerSchema.cfg`.

The following table describes the columns in the config.properties table.

Table 221. config.properties database table

Column name	Constraints	Data type	Description
CheckOutOfOrderOid		Boolean integer	If this value is 1, the SNMP Helper stops retrieving data if the returned OIDs are out of order. This is the default setting. If you want the SNMP Helper to continue to retrieve data from non-contiguous indexes, that is, to step over gaps in the returned data, set this to 0.
CheckPollDataValueRange		Boolean integer	If this value is 1, the values to be inserted into the pollData.value field are are checked to make sure that they are valid 32-bit signed integers. Set this value to 0 to bypass this check. Bypass this check only if you know that the output of the relevant polls is not likely to cause problems, and the ncpolldata schema is able to handle the output.
DefaultGetBulkMaxReps		Integer	This property defines the number assigned to the `max-repetitions` field in GetBulk requests issued by Network Manager processes. The value 20 is used when the GetBulk request contains a single varbind. If multiple varbinds are included, then the value is adjusted accordingly (divided by the number of varbinds), so that responses always contain a similar number of varbinds.
DiscoverInitialAccess		Boolean integer	If set to 1 (the default), the ncp_poller process tests SNMP credentials when it starts. Set to 0 to bypass this test.

Table 221. config.properties database table (continued)

Column name	Constraints	Data type	Description
LogAccessCredentials		Boolean integer	Controls whether SNMP access credentials (community strings and passwords) appear in plain text. If this is set to false then these strings are replaced with a fixed length string of asterisks. Default is False.
ManagedStatusUpdateInterval		Integer	Interval in seconds at which the ncp_poller process scans the NCIM managedStatus table for changes. This is the maximum time the poller should take to react to changes in managed status made in any of the following GUIs: Network Views, Network Hop View, Structure Browser. Default is 30 seconds.
PolicyUpdateInterval		Integer	Interval in seconds at which the ncp_poller process scans the ncmonitor database for changes to poll policy configuration. Default is 30 seconds.
PollerProfiling		Boolean integer	SNMP and ICMP profiling is a collection of data about the various SNMP and ICMP operations being performed by the Polling engine, ncp_poller. This profiling data is collected at a very low level in ncp_poller and therefore results in performance issue when ncp_poller is processing a heavy SNMP load; therefore profiling is turned off by default. Takes the following values: • 0: Default value. Poller profiling is off. • 1: Poller profiling is on. **Warning:** Do not change the default setting unless advised to do so by IBM Support.

Table 221. config.properties database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
UseGetBulk		Boolean integer	By default, GetBulk is not used. Set this parameter to one of the following values: • 0: Default value. GetBulk is not used by the SNMP Helper. • 1: Set this value to configure the SNMP Helper to use GetBulk requests in place of GetNext requests when SNMPv2 or v3 is used.

config.failover table

The config.failover provides the option to configure failover settings for polling.

The following table describes the columns in the config.failover table.

Table 222. config.failover database table			
Column name	**Constraints**	**Data type**	**Description**
FailedOver	Not NULL	Boolean integer	Used to facilitate failover in high availability deployments. This value must never be modified. You can check this value to determine whether the system has failed over. This field can take the following values: • 0: poller in the primary domain is actively polling, and the backup poller is on standby • 1: primary poller is not actively polling, and the backup poller has taken over
ReadyState	Not NULL	Integer	ReadyState is for **ncp_ctrl**. It is an internal state field used to determine when the poller process should start sending heartbeats. 0 is the initial value on startup, 1 is an internal intermediate state, and 2 is ready when the process starts transmitting heartbeats. ReadyState on the Backup server changes only when the server becomes active and all policies started. Until then it will be set to 0.

config.realTimeControl table

The config.realTimeControl provides the option to configure settings for the managing real-time poll policies in the MIB Grapher.

The following table describes the columns in the config.realTimeControl table. This table is used by the MIB Grapher application to maintain real-time poll policies. Although the table is not of general use, it can be used to debug MIB graphs if a problem is encountered.

Table 223. config.realTimeControl database table			
Column name	**Constraints**	**Data type**	**Description**
POLICYID	Not NULL Primary key	Integer	If there are any real-time graphs active, a record will exist for each one in this table, corresponding to the poll policy created for each graph and referenced using this POLICYID field.
HEARTBEATCOUNT	Not NULL	Integer	Provides an indication of how long the graph has been active. This value represents the number of times the graph has updated the record.
CHANGETIME		Timestamp	UNIX timestamp indicating the last time a 'heartbeat' was received.

config.tableMonitor table

The config.tableMonitor table stores data that is used to monitor the raw and historical poll data tables in the NCPOLLDATA database.

The following table describes the columns in the config.tableMonitor table.

Table 224. config.tableMonitor database table			
Column name	**Constraints**	**Data type**	**Description**
MAXPOLLDATARATE	NOT NULL	LONG64	Defines the maximum insertion rate to the raw poll data table NCPOLLDATA.polldata in units of rows of data per hour. By default, this value is 20,000,000 (20 million) rows per hour.
MAXDAILYDATAAGE	NOT NULL	LONG64	Defines the maximum age, in hours, for the data in the daily summary poll data table NCPOLLDATA.pdEwmaForDay. Data in this table is capped at 24 hours. If the age of data in this table exceeds the limit in the MAXDAILYDATAAGE field, then the Polling engine, ncp_poller logs a message and issues an alert. By default, this value is 25.
MAXWEEKLYDATAAGE	NOT NULL	LONG64	Defines the maximum age, in days, for the data in the daily summary poll data table NCPOLLDATA.pdEwmaForWeek. Data in this table is capped at 7 days. If the age of data in this table exceeds the limit in the MAXWEEKLYDATAAGE field, then the Polling engine, ncp_poller logs a message and issues an alert. By default, this value is 8.

Table 224. config.tableMonitor database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
MAXMONTHLYDATAAGE	NOT NULL	LONG64	Defines the maximum age, in days, for the data in the daily summary poll data table NCPOLLDATA.pdEwmaForMonth. Data in this table is capped at 30 days. If the age of data in this table exceeds the limit in the MAXMONTHLYDATAAGE field, then the Polling engine, ncp_poller logs a message and issues an alert. By default, this value is 32.
MAXYEARLYDATAAGE	NOT NULL	LONG64	Defines the maximum age, in days, for the data in the daily summary poll data table NCPOLLDATA.pdEwmaForYear. Data in this table is capped at 365 days. If the age of data in this table exceeds the limit in the MAXYEARLYDATAAGE field, then the Polling engine, ncp_poller logs a message and issues an alert. By default, this value is 395.

profiling database for polling

The profiling database is used by the polling engine, ncp_poller, for a variety of purposes, including the storage of summary information for poll policies and poll definitions, ping and SNMP response statistics, and general profiling statistics.

The profiling database for polling is defined in `$NCHOME/etc/precision/NcPollerSchema.cfg`.

The profiling database has the following tables:

- profiling.policy
- profiling.icmp
- profiling.snmp
- profiling.engine

profiling.policy table

The profiling.policy table provides summary information for poll policies and poll definitions.

The following table describes the columns in the profiling.policy table.

Table 225. profiling.policy database table			
Column name	**Constraints**	**Data type**	**Description**
AVGSCOPETIME	Not NULL	Integer	The average time taken to evaluate the scope of each poll (not counting the first poll).
FIRSTSCOPETIME	Not NULL	Integer	Time taken, in CPU clock ticks, for the list of entities in scope for the poll to be evaluated for the first time.
ENTITYCOUNT	Not NULL	Integer	Number of entities being monitored by this poll policy and poll definition combination.

Table 225. profiling.policy database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
POLICYID	Not NULL Primary key	Integer	Value of the ncmonitor.poll.policyId field.
POLICYNAME	Not NULL	Text	Value of the ncmonitor.policy.policyName field.
SCOPETIME	Not NULL	Integer	The total time taken, in CPU clock ticks, for the list of entities in scope for the poll to be evaluated, excluding the first time.
SCOPECOUNT	Not NULL	Integer	The number of times that the scope of the poll has been evaluated.
TARGETCOUNT	Not NULL	Integer	Number of addresses being polled by this poll policy and poll definition combination.
TEMPLATEID	Not NULL Primary key	Integer	Value of the ncmonitor.poll.templateId field.

profiling.icmp table

The profiling.icmp table stores information on ping response statistics.

The following table describes the columns in the profiling.icmp table.

Table 226. profiling.icmp database table			
Column name	**Constraints**	**Data type**	**Description**
IPVERSION	Not NULL	Text	IPv4, IPv6, or all versions.
TIMEOUTS	Not NULL	Integer	Number of ICMP requests for which no replay has been received.
PACKETSIN	Not NULL	Text	Number of ICMP packets received.
ERRORSIN	Not NULL	Integer	Number of ICMP errors received.
PACKETSOUT	Not NULL	Integer	Number of ICMP packets sent.
ERRORSOUT	Not NULL	Integer	Total number of errors encountered when sending ICMP packets.

profiling.snmp table

The profiling.snmp table stores information on SNMP response statistics.

The following table describes the columns in the profiling.snmp table.

Table 227. profiling.snmp database table			
Column name	**Constraints**	**Data type**	**Description**
ATTRIBUTESIN	Not NULL	Integer	Total number of SNMP errors received.

Table 227. profiling.snmp database table (continued)			
Column name	**Constraints**	**Data type**	**Description**
BACKOFFS	Not NULL	Integer	Total number of times exponential backoff was initiated.
DROPS	Not NULL	Integer	Total number of packets received that were not processed.
ERRORSOUT	Not NULL	Integer	Total number of tooBig errors received.
GETOPERATIONS	Not NULL	Text	Number of SNMP Get operations performed.
GETBULKSOUT	Not NULL	Integer	Total number of SNMP Get Bulk requests sent.
IPADDR	Not NULL	Text	Management IP address of the target device.
GETNEXTOUT	Not NULL	Integer	Total number of SNMP Get Next requests sent.
GETSOUT	Not NULL	Integer	Total number of SNMP Get requests sent.
NOSUCHNAMESIN	Not NULL	Integer	Total number of noSuchName errors received.
PACKETSIN	Not NULL	Integer	Total number of SNMP packets received, including errors.
PACKETSOUT	Not NULL	Integer	Total number of SNMP packets sent.
RETRIES	Not NULL	Integer	Total number of retries.
SETERRORSIN	Not NULL	Integer	Number of errors received from Set requests.
SETOPERATIONS	Not NULL	Integer	Number of SNMP Set operations performed.
SETSOUT	Not NULL	Integer	Total number of Set requests sent.
TIMEOUTS	Not NULL	Integer	Total number of SNMP operations that timed out.
TOOBIGSIN	Not NULL	Integer	Number of tooBig errors received.
WALKOPERATIONS	Not NULL	Integer	Number of SNMP Walk operations performed.

profiling.engine table

The profiling.engine table stores general profiling statistics information.

The following table describes the columns in the profiling.engine table.

Table 228. profiling.engine database table

Column name	Constraints	Data type	Description
STARTTIME	Not NULL	Timestamp	Time when profiling started.
LASTUPDATE	Not NULL	Timestamp	Last time profiling statistics were updated.
THREADSINUSE	Not NULL	Integer	Number of active threads in the core polling engine.
BATCHESQUEUED	Not NULL	Integer	Number of batches that should be running but for which there are no threads available.
AVGBATCHTIME	Not NULL	Integer	Average time in milliseconds to process each batch of work.

Chapter 14. Event enrichment databases

Use this information to understand the structure of databases used for event enrichment and for the Event Gateway plug-ins.

ncp_g_event database

The Event Gateway database enables ncp_g_event, the Event Gateway, to transfer data between Network Manager and Tivoli Netcool/OMNIbus.

The ncp_g_event database has the following database schema: config

The default configuration of the gateway is used for most systems. You can make adjustments to the configuration settings by modifying the values inserted into the Event Gateway config database. This database contains the configuration settings that define the operation of the Event Gateway. For example, you can modify the mappings used between Network Manager and Tivoli Netcool/OMNIbus and the filters that determine which events are processed.

Entity data used by the Event Gateway is stored in NCIM cache, which is a copy of the NCIM topology database. For more information on NCIM cache see the *IBM Tivoli Network Manager Reference*.

For information about ncp_g_event command-line options, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

The config database schema

The config database is used to configure event mapping between Tivoli Netcool/OMNIbus and Network Manager.

The config database can also be used to define filters that limit the number of events passed between Tivoli Netcool/OMNIbus and Network Manager.

The table below summarizes the config database schema. This schema is defined in `NCHOME/etc/precision/EventGatewaySchema.cfg`. You can specify domain-specific versions of this file using the format: `NCHOME/etc/precision/EventGatewaySchema.`*`domain_name`*`.cfg`, where *domain_name* is the name of your domain; for example, `NCHOME/etc/precision/EventGatewaySchema.NCOMS.cfg`.

Table 229. config database summary

Database name	`config`
Defined in	`NCHOME/etc/precision/EventGatewaySchema.cfg`

Fully qualified database table names	`config.defaults`
	`config.eventMaps`
	`config.failover`
	`config.nco2ncp`
	`config.ncp2nco`
	`config.precedence`

The topics below describe the database tables of the config database.

config.defaults table

The config.defaults table contains general configuration data for the Event Gateway.

The table below describes the config.defaults table.

Note: The fields `NcoAuthUserName` and `NcoAuthPassword` are now configured in the `$NCHOME/etc/precision/NcoLogins.DOMAIN.cfg file`.

Table 230. config.defaults Table Description

Column Name	Constraints	Data Type	Description
`IDUCFlushTime`	NOT NULL	Integer	Specifies the interval, in seconds, between Insert Delete Update Control (IDUC) flushes from the ObjectServer. The default value is 5.
`NcimHandleCount`		Integer	Maximum number of connections to the NCIM database server. A single handle (`NcimHandleCount = 1`) is usually sufficient, unless heavily customized stitchers are used.
`NcpServerEntity`	NOT NULL	Text	Specifies the IP address of the polling station. By default, the gateway assumes that the polling station for Network Manager is running on the local host. If you want to set a different polling station, specify the IP address of the polling station in the NcpServerEntity field. **Note:** Root cause analysis (RCA) cannot perform isolated suppression if the device specified in NcpServerEntity is not present and connected within the topology.
`ObjectServerUpdate Interval`	NOT NULL	Integer	Specifies the interval that the Event Gateway uses to queue event enrichment updates to the ObjectServer.
Fix Pack 7 backupDiscoveryCaches	NOT NULL	Integer	Specifies whether to back up the discovery cache files. A value of `0` disables backup, and `1` enables backup. The default value is `0`.
Fix Pack 7 numberOfBackupsToKeep	NOT NULL	Integer	After this number of backups have been made, older backups are deleted whenever a successful backup runs. The limit is per domain. The default value is 3.
Fix Pack 7 limitOnlyOnFullBackup	NOT NULL	Integer	Specifies which kind of backups to delete when the number of backups made exceeds the value of `numberOfBackupsToKeep`. If this parameter is set to `1`, only full discovery backups are deleted, and if it is set to `0`, backups of full and partial discoveries are deleted. The default value is `1`.

config.precedence table

The config.precedence table lists events by event ID and contains the information necessary to determine which event has precedence when multiple events occur on the same interface. Based on the event ID, the config.precedence table also determines which event map to use to process an event from Tivoli Netcool/OMNIbus.

The table below describes the config.precedence table.

Table 231. config.precedence Table Description

Column Name	Constraints	Data Type	Description
Precedence	NOT NULL	Integer	Specifies the number used by the root-cause analysis (RCA) plug-in when there are multiple events on the same entity within the network topology. The number is used to determine which of the events has precedence and therefore suppresses the other event on the interface. If a link down event has a higher Precedence value than a ping fail event, then the link down event suppresses the ping fail event on the interface. These Precedence values are unique. • 0 - An event with this Precedence value cannot suppress any other events. The event cannot become a root cause event. If the Precedence value is set to 0, then the event can become a symptom event or be marked as cause unknown. • 10000 and greater - An event with a Precedence value greater than or equal to 10000 cannot be suppressed; and the event cannot become a symptom event. The event can only become a root cause event or be marked as cause unknown.
EventMapName	NOT NULL	Text	Specifies the name of the event map from the config.eventMaps table that is used to process the event with a matching EventId.
NcoEventId	PRIMARY KEY NOT NULL	Text	Specifies the mapping from the EventId in the alerts.status table to the values of Precedence and EventMapName defined in this table. **Note:** If an event is not listed in this table, then the event is handled by the generic-ip event map.

config.eventMaps Table

The config.eventMaps table contains the event map that specifies how an event is processed. The table holds information specific to each type of Tivoli Netcool/OMNIbus event that is processed by the Event Gateway.

The table below describes the config.eventMaps table.

Table 232. config.eventMaps table description			
Column Name	**Constraints**	**Data Type**	**Description**
EventMapName	PRIMARY KEY NOT NULL	Text	Specifies the name of the event map. This value is referenced by the config.precedence table.
HandledBy		Text	An alternative to the PolledEntityStitcher, and provided for backwards compatibility. Some legacy gateway eventMaps are redundant, but removing them completely would create upgrade problems. This field allows a legacy eventMap to be mapped to a new eventMap, and behave as if the event had been handled by that eventMap.
IsPollingEvent		Integer	Specifies whether the event is a polling event. If it is a polling event then the isolated suppression RCA rule will fire for it. Defaults to a value of 1 (True).
EventCanFlap		Boolean	Indicates if it is possible for the event to flap. Flapping is a condition where a device or interface connects to and then disconnects from the network repeatedly in a short space of time. This causes problem and clear events to be received one after the other for the same device or interface. Setting the `EventCanFlap = 1` informs the RCA plug-in of this condition. The RCA plug-in places these events in the mojo.events database with `TimedEscalation = 1` and are left there for 30 seconds. After 30 seconds one of the RCA plug-in stitchers processes all events that are at least 30 seconds old and have the `TimedEscalation = 1` setting. By waiting 30 seconds to process the event, the system ensures that the entity that generated the event has settled down and is not flapping.

config.nco2ncp table

The config.nco2ncp table is used to filter events being passed from Tivoli Netcool/OMNIbus to Network Manager.

The table below describes the config.nco2ncp table.

Table 233. config.nco2ncp table description			
Column Name	**Constraints**	**Data Type**	**Description**
EventFilter	NOT NULL	Text	Specifies a filter that indicates which events should be processed by the Event Gateway. Events that match the filter are processed. **Attention:** Do not modify this filter unless you are aware of the consequences of the modification. Only advanced users should modify this filter.
StandbyEvent Filter		Text	Used when the primary server is down and the backup server is active. The standby filter only allows ItnmHealthCheck events through the Event Gateway. These events are passed to the Failover plugin and tell the system to switch back to primary mode.
FieldFilter	Externally defined vblist data type	Object	Specifies a subset of alerts.status fields that are passed through to the Event Gateway. If the field filter is empty then all alerts.status fields are are passed through. The purpose of this filter is to limit the fields passed through to the minimum required set in order to lighten the processing load.

The gateway determines whether to insert a new record or update an existing record according to whether the ObjectServer sends the event as an insert using IDUC or as an update.

config.ncp2nco table

The config.ncp2nco table is used to filter and map events passed from Network Manager to Tivoli Netcool/OMNIbus.

The table below describes the config.ncp2nco table.

Table 234. config.ncp2nco table description			
Column Name	**Constraints**	**Data Type**	**Description**
FieldFilter	Externally defined vblist data type	Object	Specifies the set of ObjectServer fields that may be updated by the Event Gateway.

config.failover table

The config.failover table contains the failover configuration and current failover state of the Event Gateway component.

Attention: Do not manually change the values of the config.failover table. In a failover configuration, the FailedOver field is modified by the virtual domain process.

The table below describes the config.failover table.

Table 235. config.failover table description			
Column Name	**Constraints**	**Data Type**	**Description**
FailedOver	NOT NULL	Boolean	Specifies the failover state. • 0 - Not in a failover state • 1 - In a failover state
ReadyState	ReadyState	Integer	ReadyState is for **ncp_ctrl**. It is an internal state field used to determine when the poller process should start sending heartbeats. 0 is the initial value on startup, 1 is an internal intermediate state, and 2 is ready when the process starts transmitting heartbeats. ReadyState on the Backup server changes only when the server becomes active and all policies started. Until then it will be set to 0.

ncp_g_event plug-in databases

The Event Gateway plug-in database tables are used by the plug-ins to store processing data.

RCA plug-in database

The RCA plug-in database tables enable the RCA plug-in to perform root-cause analysis.

mojo.events events database table

The mojo database stores all event records sent for root cause analysis by the Event Gateway. The database contains the mojo.events table.

The mojo database is defined in NCHOME/etc/precision/RCASchema.cfg.

The column names of the records are used in many of the conditional filters when constructing event correlation methods.

The table below describes columns in the mojo.events table.

Table 236. Descriptions for the mojo.events database table columns			
Column Name	**Constraints**	**Data type**	**Description**
ChangeTime	TIMESTAMP Not null	Long Integer	Specifies the time the event was last updated by the RCA plug-in.
CreateTime	TIMESTAMP Not null	Long Integer	Specifies the time the event was first seen by the RCA plug-in.
Description		Text	Specifies a textual description of the event.
EntityType	Not null	Int	A value of 1 or 8 indicates that this is a chassis device.
EventId		Text	Type of event; for example NmosPingFail.

Table 236. Descriptions for the mojo.events database table columns (continued)

Column Name	Constraints	Data type	Description
FirstOccurrence	TIMESTAMP Not null		Time the event was first seen by Tivoli Netcool/OMNIbus. **Note:** This value is set by the probe, not by Tivoli Netcool/OMNIbus. This means that this field arrives at the ObjectServer with a value already set. Tivoli Netcool/OMNIbus never touches this field.
IsIsolationPoint	Not null	Int	Can take the following values: • 0 - No • 1 - Yes
IsLoopbackInterface	Not null	Int	Can take the following values: • 0 - No • 1 - Yes
IsMasterEvent	Not null	Int	Can take the following values: • 0 - This is *not* the master event on the entity. • 1 - This is the master event on the entity. **Note:** The master event on an entity will suppress all other events on that entity, should there be any.
IsOrphan	Not null	Int	Can take the following values: • 0 - No • 1 - Yes **Note:** This field is used internally to enable the RCA plug-in to reprocess suppressed events whose root cause event has since been deleted.
LastOccurrence	TIMESTAMP Not null	Long Integer	Time the event was last seen by Tivoli Netcool/OMNIbus. **Note:** This value is set by Tivoli Netcool/OMNIbus itself when it receives the event.
NmosCauseType	Not null	Int	Can take the following values: • 0 - Unknown • 1 - Root cause • 2 - Symptom • 3 - Not suppressing and not suppressed
NmosEntityId	Not null	Int	Entity on which the event occurred.
NmosManagedStatus	Not null	Int	Managed status of the entity.

Table 236. Descriptions for the mojo.events database table columns (continued)

Column Name	Constraints	Data type	Description
NmosObjInst	Not null	Int	Entity ID for the chassis related to the entity on which the event occurred.
NmosSerial	Not null	Int	Serial number of the event that suppressed this event.
Precedence		Int	A value from 0 to 10,000 indicating, where there are multiple events on the same entity, the event to be used to suppress the other events on that entity. The event with the highest precedence value suppresses the others.
RemoteNodeAlias		Text	Network address of the remote network entity
Serial	Primary key Not null	Uint	Serial number of this event in Tivoli Netcool/OMNIbus. Used to uniquely identify the event and the record in mojo.events.
Severity	Not null	Int	Severity of the event.
State	Not null	Int	Event state for this event.
SuppressionState	Not null	Int	Suppression state for this event. This field can take the following values: • 0 - No suppression • 1 - Entity suppression • 2 - Contained suppression • 3 - ConnectedSuppression • 4 - IsolatedSuppression • 5 - PeerSuppression
SuppressionTime	TIMESTAMP Not null	Long Integer	Time the event was last suppressed.
TimedEscalation	Not null	Int	Can take the following values: • 0 - Tells RCA plug-in to process the event immediately. • 1 - Tells RCA plug-in to process the event after 30 seconds. This is usually set by events that can flap. • 2 - Set for events that previously had the `TimedEscalation = 1` setting and have since been processed.

config.defaults database table

The config.defaults database table stores configuration data for the RCA plug-in event queue.

The `config` database is defined in `NCHOME/etc/precision/RCASchema.cfg`.

The table below describes columns in the config.defaults database table.

Table 237. Descriptions for the config.defaults database table columns

Column Name	Constraints	Data type	Description
RequeueableEventIds		Text	Specifies that events of certain types can be requeued if the RCA plug-in queue becomes very large. This ensures that only one event of a specific event type exists in the queue at any one time.
MaxAgeDifference	NOT NULL	Text	Specifies the maximum age difference between events that pass through the RCA plug-in. Events that have a difference in age greater than this specified value cannot suppress each other. By default this option is switched off, that is, set to 0. This means that events on the same entity suppress each other regardless of the age of the events.
HonourManagedStatus		Integer Boolean	Specifies whether the RCA plugin uses the managed status of an entity in calculating the root cause of an event. If this value is set to 1, then events from unmanaged devices are ignored.
GraphTopologyNames		List	Specifies topologies that are to be used to create the RCA graph. By default the topologies used are the following: • RelatedToTopology • LocalVlanTopology
TopologyChangesThreshold	NOT NULL	Integer	Maximum number of changes allowed before deleting the graph and recreating from scratch. If the current number of changes is less than this threshold value, the dynamic updates are made to the graph. Default value is 100.

SAE plug-in database

The SAE plug-in database tables enable the SAE plug-into generate service-affected events for services such as MPLS VPNs and IP paths.

The table below summarizes the config database schema.

Table 238. config database summary

Database name	`config`

Defined in	`NCHOME/etc/precision/SaeSchema.cfg`
	`NCHOME/etc/precision/SaeCluster.cfg`
	`NCHOME/etc/precision/SaeIPPath.cfg`
	`NCHOME/etc/precision/SaeItnmService.cfg`
	`NCHOME/etc/precision/SaeMplsVpn.cfg`

Table 238. config database summary (continued)	
Fully qualified database table names	`config.serviceTypes`

config.serviceTypes table

The config.serviceTypes table contains configuration information for the SAE plug-in.

The table below describes columns in the config.serviceTypes table.

Table 239. Descriptions for the config.serviceTypes database table columns

Column Name	Constraints	Data type	Description
ServiceTypeName	Primary key Not null	Text	Represents the type of service; for example, "MPLS VPN Edge Service" or "IP Path". This string will appear in the eventId field of the SAE event in the ObjectServer and will also form part of the Summary field of the SAE event in the ObjectServer.
CollectionEntityType	Not null	Integer	Used to specify the entity type that corresponds to the collection that the SAE will be generated for; for example 17 (VPN), 34 (ITNM Service), or 80 (IP Path). For a listing of possible entity types in the NCIM entityType table, see the *IBM Tivoli Network Manager Reference*.
ConstraintFilter	Optional	Text	Used to constrain the entries of interest in the collection table, if necessary. For example, for the table networkVpn, entries that have Type = 'MPLS Core' are excluded. In this case, the contraint filter is formulated as follows: "networkVpn->VPNTYPE <> 'MPLS Core'"
CustomerNameField	Optional	Text	Used to specify where to obtain the customer name string to append to the Summary field of the event in the ObjectServer. For example, if the ServiceEntity record contains the field Customer then this can be used to retrieve a string such as "ACME Inc" from the ServiceEntity topology record. This example would take the form "entityData->DESCRIPTION".

Related reference

entityType
The entityType table provides a comprehensive list of every entity type in NCIM. It belongs to the category *entities*.

ncp_g_event plug-in database tables in ncmonitor

Use this information to understand which Event Gateway configuration tables are available in the ncmonitor database and what type of information each table contains. Most of these tables relate to Event Gateway plug-in configuration.

The table below lists each Event Gateway plug-in configuration tables in the ncmonitor database and explains the purpose of the table.

Table 240. Event Gateway plug-in configuration tables in ncmonitor	
Table	**Description**
ncmonitor. gwPluginTypes	Lists the available plugin libraries. Similar idea to poller templates/poll definitions. This should contain a single entry for the Adaptive Polling plugin.
ncmonitor. gwPlugins	Lists the plugins that are enabled. Similar idea to the poller policies. This should contain a handful of entries for the Adaptive Polling plugin.
ncmonitor. gwPluginEventMaps	Identifies the event maps that each plugin is interested in. Plugins are only supplied with events handled by listed event maps.
ncmonitor. gwPluginEventStates	Identifies the type of event that each plugin is interested in. Plugins are only supplied with events handled by listed event states.
ncmonitor. gwSchemaFiles	Lists the OQL schema files to be read by the Event Gateway. These files are read in the order they are listed, and therefore the EventGatewaySchema file should be the first file listed, as it defines the tables. By default, this lists the EventGatewaySchema file. The purpose of this table is to allow additional self-contained schema files to be supplied for future device support.
ncmonitor. gwPluginConf	Allows optional configuration variables to be defined for specific plugins. **Note:** It is intended for values that would not, under normal circumstances, be changed.

Chapter 15. ncp_class database

The ncp_class database enables the Active Object Class (AOC) manager, ncp_class, to manage AOCs.

The CLASS database consists of two main tables, activeClasses and staticClasses, which are populated at startup as CLASS reads the AOC definitions. The two tables represent two different views of the class hierarchy.

The ncp_class database is described in the following file.

`$NCHOME/etc/precision/ClassSchema.cfg`

Note: For information about ncp_class command-line options, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

class.activeClasses table

The class.activeClasses table holds the full definition of every Active Object Class (AOC).

The table below describes the activeClasses table.

Table 241. class.activeClasses Database Table Schema

Column name	Constraints	Data type	Description
`ClassName`	PRIMARY KEY NOT NULL UNIQUE	Text	Specifies the unique name of the AOC.
`ClassId`	PRIMARY KEY NOT NULL	Integer	Specifies an identifier for the AOC.
`SuperClass`	NOT NULL	Text	Specifies the name of the parent AOC.
`Dictionary`		Text	List of dictionaries.
`Instantiate`	NOT NULL	Text	Specifies the rules for instantiating the AOC.
`VisualIcon`	NOT NULL	Text	Specifies the icon associated with the AOC, which is displayed in a user interface.
`MenuRules`		List type object	List of executable rules.
`Menu`		List type object	Menu for the AOC.
`ActionType`	Externally defined actions data type	Integer	Specifies the list of actions to be taken.

class.staticClasses table

The class.staticClasses table holds the contents of a raw Active Object Class (AOC) as it is defined in the `.aoc` file.

The table below describes the class.staticClasses table.

Table 242. class.staticClasses Database Table Schema

Column name	Constraints	Data type	Description
`ClassName`	PRIMARY KEY NOT NULL UNIQUE	Text	Specifies the unique name of the AOC.
`ClassId`	PRIMARY KEY NOT NULL	Integer	Specifies an identifier for the AOC.
`SuperClass`	NOT NULL	Text	Specifies the name of the parent AOC.
`Dictionary`		Text	List of dictionaries.
`Instantiate`	NOT NULL	Text	Specifies the rules for instantiating the AOC.
`Extensions`	Externally defined extension data type	Object of extension data type	Specifies the list of extensions contained within the AOC.
`VisualIcon`	NOT NULL	Text	Specifies the icon associated with the AOC.
`MenuRules`		List type object	List of executable rules.
`Menu`		List type object	Menu for the AOC.
`ActionType`	Externally defined actions data type	Integer	Specifies a list of actions to be taken.

class.classIds table

The class.classIds table stores the lookup values for class IDs from class name.

The table below describes the `classIds` table.

Table 243. class.classIds Database Table Schema

Column name	Constraints	Data type	Description
`ClassId`	PRIMARY KEY NOT NULL UNIQUE	Integer	Specifies the numerical ID for the Active Object Class (AOC).

Table 243. class.classIds Database Table Schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
`ClassName`	PRIMARY KEY NOT NULL UNIQUE	Text	Specifies the unique name of the AOC.

Chapter 16. ncp_ctrl database

The ncp_ctrl database enables the master process controller, ncp_ctrl, to manage and monitor the status of Network Manager processes.

The ncp_ctrl database is defined in the `$NCHOME/etc/precision/CtrlSchema.cfg` file.

The services.config Table

The `services.config` database table is an active table for managed processes.

This table is used for the ncp_ctrl process when it overwrites trace files. To enable and configure trace file rotation, do not change this database; use the log file rotation environment variables.

The table below describes the services.config database table.

Table 244. services.config database table schema

Column name	Constraints	Data type	Description
enableTraceFileRotation	NOT NULL	Integer	Specifies whether log file rotation is enabled.

The services.inTray Table

The `services.inTray` database table is an active table for managed processes.

Data is inserted into this table by ncp_ctrl when it reads its configuration file. Processes with an entry in this database are started by ncp_ctrl as managed processes. If an entry is removed from this table, it is stopped by ncp_ctrl.. The ncp_ctrl process also monitors the status of processes named in the `inTray` table and restarts them if they are stopped.

The table below describes the services.inTray database table.

Table 245. services.inTray Database Table Schema

Column name	Constraints	Data type	Description
argList		List of type text	Specifies a list of arguments sent to the service.
binaryName	NOT NULL	Text	Base name for the binary that implements the service. This is used in preference to the serviceName field to launch the service.
dependsOn		List of type text	Specifies a list of processes, prerequisites, required to run the current services.
domainName		Text	Specifies the domain under which the service is running.
hostName		Text	Specifies the name of the system upon which the service is running.
interval		Integer	Specifies the average interval between heartbeat signals from the service.

Table 245. services.inTray Database Table Schema (continued)

Column name	Constraints	Data type	Description
logFile		Text	Specifies the name of file in which output is logged.
logLevel		Text	The logging level for the process. By default, this field has the value `warn`. The value in this table updates dynamically if the logging level is changed for the process on the command line. Changing this value in the table dynamically changes the logging level for the process even if it is already running.
processId		Long integer	Specifies the process ID of the service.
memUsage		Long integer	The native memory of the process.
retryCount		Integer	Specifies the number of times to attempt to restart a service. **Note:** Providing the value 0 (zero) means no attempt is made to restart a service when it stops, while -1 sets the service to always start again when it stops.
serviceId	PRIMARY KEY NOT NULL UNIQUE	Integer	Specifies the unique ID of the service and is assigned internally.
serviceKey	NOT NULL UNIQUE	Text	Auto-generated unique key for a process.
serviceName	NOT NULL	Text	Specifies the name of the service to be managed.
servicePath		Text	Specifies the full path to the service, which might be `NCHOME/precision/platform/$PLATFORM/bin` if NULL.

Table 245. services.inTray Database Table Schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
serviceState	Externally defined serviceState data type	Integer	Specifies an integer which reflects the current operational state of the service. 0 - The service is alive but currently idle. 1 - The service is waiting for its prerequisites, that is, waiting for its dependencies to be satisfied. 2 - The service is waiting to begin sending heartbeats. 3 - The application is currently starting, which is Fork service. 4 - The service is alive and running. 5 - The service is not functioning correctly. 6 - The service is stopped. 7 - The service has failed. 8 - The service is shutdown.
traceLevel		Integer	The trace level for the process. By default, this field has the value 0 (zero). The value in this table updates dynamically if the trace level is changed for the process on the command line. Changing this value in the table dynamically changes the trace level for the process even if it is already running.

The services.slaveCtrl Table

The `services.slaveCtrl` table is populated automatically and serves as a reference for other instances of the ncp_ctrl process that are running in slave mode.

The table below describes the services.slaveCtrl database table.

Important: Do not use the OQL Service Provider to manually insert records into this table.

Table 246. services.slaveCtrl Database Table Schema			
Column name	**Constraints**	**Data type**	**Description**
`slaveId`	UNIQUE PRIMARY KEY NOT NULL	Integer	Specifies the unique ID number of the subordinate ncp_ctrl process running in slave mode.
`domainName`		Text	Specifies the domain under which the subordinate ncp_ctrl process is operating.
`hostName`		Text	Specifies the name of the system on which the subordinate ncp_ctrl process is running.

Table 246. services.slaveCtrl Database Table Schema (continued)			
Column name	**Constraints**	**Data type**	**Description**
processId		Long integer	Specifies the process ID of the subordinate ncp_ctrl process.
slaveState	Externally defined serviceState data type	Integer	Specifies the current operational state of the subordinate process.

The services.unControlled Table

The services.unControlled table is a read-only table used to monitor uncontrolled services.

If the ncp_ctrl process receives a heartbeat from a service but does not start, the service is considered an uncontrolled service.

Important: Do not insert the information provided into this table. If you insert the information into the unControlled table, then your system might not function correctly.

The table below describes the services.unControlled database table.

Table 247. services.unControlledDatabase Table Schema			
Column name	**Constraints**	**Data type**	**Description**
serviceId	UNIQUE PRIMARY KEY NOT NULL	Integer	Specifies the unique ID of the service.
serviceName	NOT NULL	Text	Specifies the name of the service.
domainName		Text	Specifies the domain under which the service is running.
hostName		Text	Specifies the name of the system on which the service is running.
processId		Long integer	Specifies the process ID of the service.
serviceState	Externally defined serviceState data type	Integer	Specifies an integer which reflects the current operational state of the service.
interval		Integer	Specifies the average interval between heartbeat signals from the service.

The services.unManaged Table

The services.unManaged table is used by the ncp_ctrl process to start and stop unmanaged processes. This table is also used by other Network Manager processes to instruct the ncp_ctrl process to start their subprocesses.

The ncp_disco process, for example, instructs the ncp_ctrl process to start the finders, helpers and agents by sending inserts into the services.unManaged table of the ncp_ctrl process. Inserting or deleting records from the table causes the corresponding processes to be started or stopped.

The table below describes the services.unManaged database table.

Table 248. services.unManaged Database Table Schema

Column name	Constraints	Data type	Description
argList		List of type Text	Specifies a list of arguments sent to the service process.
dependency		Long integer	Specifies the process ID of the parent that the service is dependent upon. Examine the content of this field to determine whether this unmanaged process is dependent or independent. • If this field contains a process ID value then this means that this is a *dependent* unmanaged process. A dependent unmanaged process is stopped by ncp_ctrl if the parent process dies. When this field is set then it contains the PID of the parent process. An example of dependent unmanaged processes are the discovery agents started by the parent Discovery engine, ncp_disco, process. • If this field contains a NULL value then this means that this is an *independent* unmanaged process. An independent unmanaged process continues to run if the parent process dies.
endSignal		Integer	The signal to be used to end the process. The default value is 9.
hostName		Text	Specifies the system on which the service is running.
logFile		Text	Specifies the name of the file in which output is logged.
processId		Long integer	Specifies the service process ID.
serviceId	UNIQUE PRIMARY KEY NOT NULL	Integer	Specifies the unique ID of the service.
serviceName	NOT NULL	Text	Specifies the name of the service.
servicePath		Text	Specifies the full path to the service process. If the value is set to NULL, then this default path is used: `NCHOME/precision/platform/$PLATFORM/bin`.
serviceState	Externally defined serviceState data type	Integer	Specifies an integer which reflects the current operational state of the service.

Chapter 17. ncp_trapMux database

The ncp_trapMux database enables the SNMP Trap Multiplexer, ncp_trapMux, to forward traps to multiple ports and capture and replay traps.

The ncp_trapMux database is defined in the `$NCHOME/etc/precision/TrapMuxSchema.cfg` file.

trapMux.command table

The trapMux.command database table is an active table used to control the ncp_trapmux process.

The table below describes the `trapMux.command` table.

Table 249. trapMux.command Database Table Schema

Column name	Constraints	Data type	Description
`command`	NOT NULL	Text	Specifies the command to issue to the ncp_trapmux process.
`fileName`		Text	Specifies the file on which to perform the command, if applicable.

trapMux.config table

The trapMux.config table contains the main configuration data for the ncp_trapmux process.

The table below describes the `trapMux.config` table.

Table 250. trapMux.config Database Table Schema

Column name	Constraints	Data type	Description
`port`		Integer	Specifies the port on which to listen for traps.

trapMux.sinkHosts table

The trapMux.sinkHosts table contains details of the hosts to which traps are forwarded and the port numbers

The table below describes the `trapMux.sinkHosts` table.

Table 251. trapMux.sinkHosts Database Table Schema

Column name	Constraints	Data type	Description
`host`	NOT NULL	Text	Specifies the host name or IP address to which traps are forwarded.
`port`	NOT NULL	Integer	Specifies the port number of the host name or IP address to which traps are forwarded.

Chapter 18. ncp_virtualdomain database

The ncp_virtualdomain database enables the virtual domain subsystem to support Network Manager failover.

The ncp_virtualdomain database uses two database tables: `config` and `state`. The health check status and filters are stored in these tables.

To configure the operation of virtual domains specify OQL inserts to the `VirtualDomainSchema.cfg` file. The primary server and the backup server each have a `VirtualDomainSchema.cfg` file. Ensure that you make the same changes to both configuration files.

config database schema

The config database schema is defined in the `NCHOME/etc/precision/VirtualDomainSchema.cfg` directory. The config database schema has one table: config.defaults.

The table below describes the defaults table. The defaults table holds the time periods for the failover checks.

Table 252. config.defaults table description

Column name	Constraints	Data type	Description
m_AutoTopologyDownload	NOT NULL	Integer	Specifies the network topology transferred from the primary server when the backup server starts. If this option is set, then the Network Manager server downloads the topology every time the TCP connection is lost and reestablished. • 0 - Topology is only downloaded once when the Network Manager server is started. • 1 - Topology is downloaded each time a new TCP connection is made. This is the default value.
m_FailoverTime	NOT NULL	Integer	Specifies the maximum difference between the current time and the Health Check Resolution event timestamp. If the difference exceeds this value, then Network Manager fails over. The default value is 300 seconds.
m_HealthCheckPeriod	NOT NULL	Integer	Specifies the time period between each health check. The health check applies the filters in state.filters to the values in state.services (the current state of the processes monitored by the CTRL process). The default value is 60 seconds.

Table 252. config.defaults table description (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_SocketKeepAlivePeriod	NOT NULL	Integer	Specifies the time, in seconds, between messages that Virtual Domain sends to indicate that a connection is still active. If three of these messages are missed, the connection is considered inactive and Virtual Domain tries to open a new connection. By default this is 60, that is, one minute.

state database schema

The state database schema is defined in the `NCHOME/etc/precision/VirtualDomainSchema.cfg` directory. The state database schema has three tables: state.services, state.domains, and state.filters.

The services table holds the status of the processes monitored by the CTRL process. The table below describes the columns of the services table.

Table 253. state.services Table Descriptions			
Column name	**Constraints**	**Data type**	**Description**
m_ArgList		List	Specifies a list of arguments sent to the service process.
m_ChangeTime	NOT NULL	Timestamp	Specifies a timestamp from the last time the status of this service was updated by the CTRL process.
m_CtrlState	NOT NULL	Integer	Specifies an integer which reflects the operational state of the service. • 0 - The service is alive but idle. • 1 - The service is waiting for its prerequisites, that is, waiting for its dependencies to be satisfied. • 2 - The service is waiting for a heartbeat from another service. • 3 - The application is starting, that is, Fork service. • 4 - The service is alive and running. • 5 - The service is not functioning correctly. • 6 - The service is stopped. • 7 - The service failed. • 8 - The service is shutdown.
m_Domain	NOT NULL	Text	Specifies the domain name under which the service is running.
m_ExtraInfo		Vblist	Specifies additional information. The default value is empty.
m_Pid	NOT NULL	Integer	Specifies the process ID for the component.

Table 253. state.services Table Descriptions (continued)			
Column name	**Constraints**	**Data type**	**Description**
m_ServiceName	PRIMARY KEY NOT NULL	Text	Specifies the service name of the component monitored by the CTRL process.

The domains table holds the status of the primary and backup domains in the failover architecture. This table always contains an entry for the primary server and the backup server.

The table below describes the columns of the domains table.

Table 254. state.domains Table Descriptions			
Column name	**Constraints**	**Data type**	**Description**
m_ActingPrimary	NOT NULL	Integer	Specifies whether the Network Manager server is acting as the primary server and is monitoring the network. • 0 - Not acting as the primary server • 1 - Acting as the primary server
m_Backup	NOT NULL	Integer	Specifies whether the server is configured as the backup server. This value is automatically set by the configuration defined in the `$NCHOME/etc/precision/ConfigItnm.`*`DOMAIN`*`.cfg` file, or by inclusion of the `-primaryDomain` command-line option on components started by the CTRL process. • 0 - Not configured as the backup server • 1 - Configured as the backup server
m_ChangeTime	NOT NULL	Long	Specifies the timestamp from the last time the status of the domain was updated by the Event Gateway.
m_Domain	NOT NULL	Text	Specifies the domain in which this installation of Network Manager is running. The domain name must be different from the names of the primary and backup servers.
m_HealthStatus	NOT NULL	Integer	Specifies the status of the Health Check events. • 0 - Unhealthy. The Network Manager server generated a Health Check Problem event or the existing Health Check Resolution event exceeded the m_failover time period. • 1 - Healthy. The Network Manager server generated a Health Check Resolution event within the m_failover time period.

The filters table contains the filters that are to be applied to the values in the state.services table. The table below describes the columns of the filters table.

Table 255. state.filters Table Descriptions			
Column name	**Constraints**	**Data type**	**Description**
m_ServiceName	NOT NULL	Text	Specifies the unique name of the service to which the filter is applied.
m_Filter	NOT NULL	Text	Specifies the OQL filter to apply.
m_Description		Text	Specifies a description of the filter operation.

Network Manager provides a default filter for the following components:

There is one filter for each component except for the ncp_poller component, which instead can have a filter for each poller defined. Each filter checks that the m_CtrlState value is not set to 7 (the service has not failed), and that the timestamp m_ChangeTime is not older than 300 seconds.

Default filters for the state.filters table are provided for the components below.

- Polling engine, ncp_poller: multiple filters can be defined, one for each poller defined in the `CtrlServices.cfg` file.
- Event Gateway, ncp_g_event
- Topology manager, ncp_model

Example Virtual Domain configuration

Use these examples to familiarize yourself with the default OQL inserts that are appended to the `VirtualDomainSchema.cfg` file to configure Virtual Domain when it is launched.

Remember: Both the primary server and backup server have a `VirtualDomainSchema.cfg` configuration file. Ensure that you make identical changes to both configuration files.

Example configuration of the config.defaults table

The OQL sample below configures the config.defaults table.

```
insert into config.defaults
(
        m_HealthCheckPeriod,
        m_FailoverTime,
        m_AutoTopologyDownload
)
values
( 60, 300, 1 );
```

This insert triggers the following behavior:

- Virtual Domain performs a health check of each of the processes listed in the state.services table every 60 seconds. The health check applies the filters in the state.filters table to the values in state.services (the current state of the processes being monitored by the CTRL process).
- Virtual Domain has a default time difference of 300 seconds between the current time and the Health Check Resolution event timestamp. When the difference exceeds this value Network Manager fails over. The time difference is set to this value to avoid spurious failover.
- The backup server downloads the topology every time the TCP connection is lost and remade with the primary server.

Example configuration of the state.filters table: Event Gateway filter

The following sample OQL insert specifies one of the default filters provided with Network Manager.

```
insert into state.filters
(
        m_ServiceName,
```

```
        m_Filter,
        m_Description
)
values
(
        "ncp_g_event",
        "m_ChangeTime > eval(long,'$TIME - 300') and m_CtrlState <> 7",
        "The Gateway has been running within the last 300 seconds"
);
```

This insert triggers the following behavior:

- Virtual Domain applies the filter (to be specified later in the insert) against the Event Gateway process, ncp_g_event.
- Virtual Domain applies a filter that checks that the m_CtrlState value is not equal to 7 (the service has not failed), and that the time stamp m_ChangeTime is not older than 300 seconds.
- Virtual Domain outputs the following description of the filter: `The Gateway has been running within the last 300 seconds`.

Example configuration of the state.filters table: filter for additionally configured poller

Additionally configured pollers can trigger failover. The following sample OQL insert specifies the commented out example filter for an additionally configured poller provided in the default version of the `VirtualDomainSchema.cfg` file. Remove the slashes in order to uncomment this filter. The m_ServiceName setting in the filter below must match the serviceName column specified for the additionally configured poller in the `$NCHOME/etc/precision/CtrlServices.cfg` file.

```
// insert into state.filters
// (
//     m_ServiceName,
//     m_Filter,
//     m_Description
// )
// values
// (
//     "PingPoller",
//     "m_ChangeTime > eval(time,'$TIME - 300') and m_CtrlState <> 7",
//     "The Poller has been running within the last 300 seconds"
// );
```

This insert triggers the following behavior:

- Virtual Domain applies the filter (to be specified later in the insert) against the additionally configured poller, PingPoller.
- Virtual Domain applies a filter that checks that the m_CtrlState value is not equal to 7 (the service has not failed), and that the time stamp m_ChangeTime is not older than 300 seconds.
- Virtual Domain outputs the following description of the filter: `The Poller has been running within the last 300 seconds`.

Chapter 19. NCIM topology database

The Network Connectivity and Inventory Model (NCIM) topology database is a relational database that Network Manager uses to consolidate topology data about the physical and logical composition of devices, layer 1, layer 2, and 3 connectivity, routing protocols and network technologies such as OSPF, BGP and MPLS Layer 3 VPNs.

Usage considerations

This information about the NCIM database assumes that you are familiar with relational databases. It also assumes that you are familiar with SQL query techniques used to extract data from relational databases. You can use these query techniques to query the NCIM database to obtain topology data. Expert users can manipulate data in the NCIM database using insert, update, and delete statements.

Related reference

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

Chapter 20. About NCIM

Use this information to understand how NCIM works, what you can use NCIM for, and the how the NCIM database is structured to store data, and support queries and extensibility.

Topology database tasks

You can use the NCIM topology database to perform the tasks, such as extracting data topology using SQL queries, exporting data from the topology database to third-party applications, and including data from third-party sources and from customized discoveries by extending the database.

Use the NCIM topology database to perform the following tasks:

Extracting data
You can write SQL queries to extract topology data from NCIM. SQL queries can be made using ncp_oql as well as using third-party tools.

Integrating with third-party applications
You can export data from the topology database to third-party applications. In order to do this, you must understand the structure of the database.

Extending the database
You can extend the database to include data from third-party sources and from customized discoveries. For example, discovery stitchers may be configured to look up customer details from a third-party source based on IP address.

Topology database architecture

Use this information to understand how the NCIM topology database works.

The following figure shows how Network Manager populates NCIM, and shows how the topology data is shared and accessed by different processes within Network Manager.

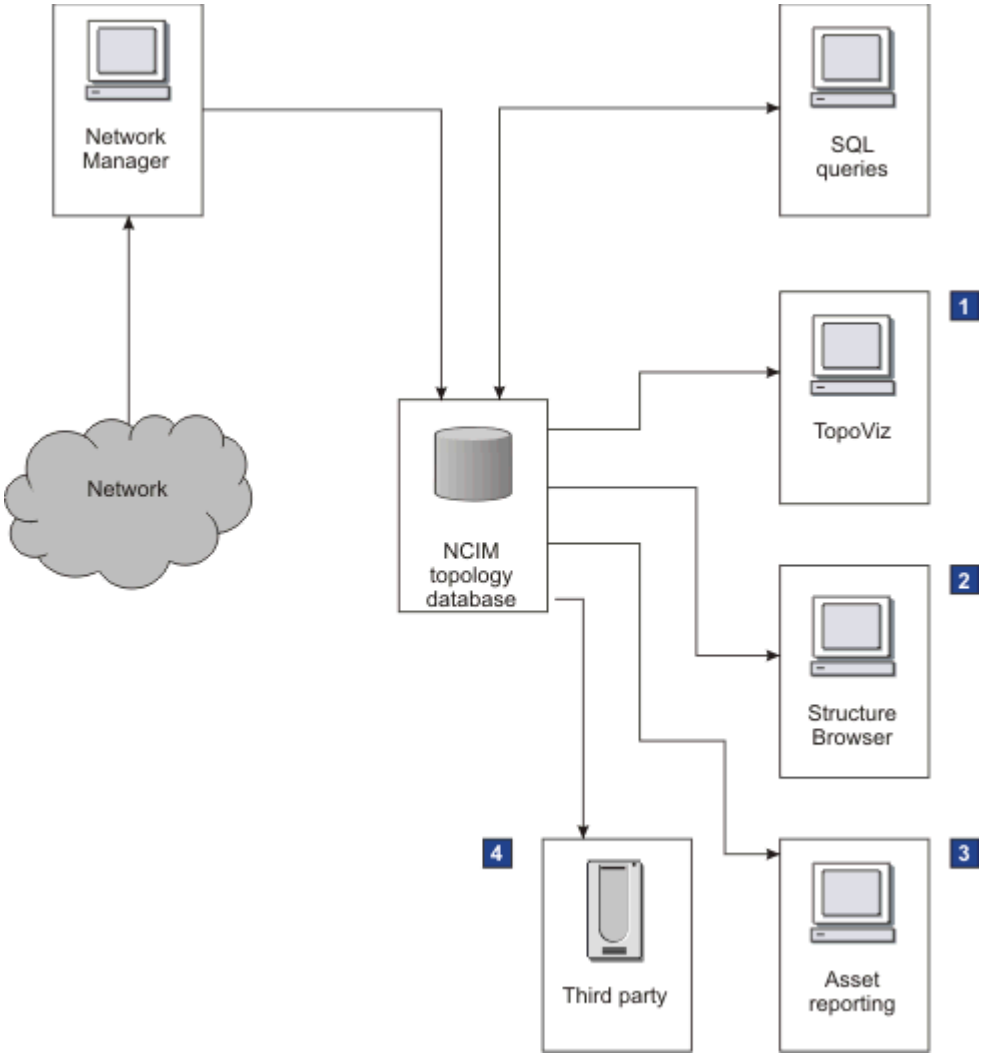


Figure 2. NCIM working with Network Manager

Network Manager populates NCIM by means of the Discovery engine, ncp_disco, and Topology Manager, ncp_model, processes.

The topology data in NCIM can be shared and accessed by the following processes and applications:

1 TopoViz
Used for displaying topology maps.

2 Structure Browser
Used for navigating within the containment structure of devices in the topology.

3 Asset reporting
Used for asset reporting software.

4 Integrations with third-party applications
Used for example provisioning software that requires regularly updated topology data from Network Manager. These activities require knowledge of programming languages such as Java and Perl.

Topology database properties

Use this information to understand how the NCIM topology database is structured to store data, and support queries and extensibility.

NCIM data storage

The NCIM relational database are divided into core tables and attribute tables. Core tables define all entities within NCIM together with the relationships between these entities. Attribute tables contain attribute data for each entity; they are specific to Network Manager.

NCIM database structure

Base information for the discovered network resources and relationships is held within the `entityData` table. Resource-specific attribute data is held in product-specific extension tables that typically have a foreign key relationship with the core-model `entityData` table.

The NCIM topology database also holds meta-data in tables such as the mappings, enumerations, CIDRInfo and deviceFunction tables. You can query this data to get useful, typically human-readable information for device attributes. For example, you can determine the user-friendly name of BGP AS numbers.

The NCIM topology database has been designed to be familiar to users who work with the MODEL database in legacy object-oriented format. This is most apparent in the naming of NCIM relational database tables and fields. Where possible, the naming is the same as that used in MODEL.

Multiple domain queries

NCIM allows multiple network domains to be stored in the database simultaneously. A domain is a scoped set of entities discovered and managed by an application, such as Network Manager.

A single SQL query on the NCIM database can extract data from multiple domains. This is in contrast to Object Query Language (OQL) queries on the Topology manager, ncp_model, topology database, which are able to extract information only from a single domain at a time.

Extensibility

The NCIM topology database can hold additional data that is collected during a customized discovery. For example, discovery stitchers can be configured to look up customer details from a third-party source based on an IP address. It is possible to configure MODEL to populate NCIM with this additional data and to configure NCIM to store this additional data in the form of key-value pairs.

Continuing the example, you might configure NCIM to store a customer name and customer type, associated with each main node entity discovered. It is also possible to modify NCIM to create new multicolumn tables and configure MODEL to populate these tables following a customized discovery. These modifications enable NCIM to store more custom data. For example, you might want to store a set of data on each customer associated with an IP address.

Related concepts

Domains

A domain is a scoped set of entities that are discovered and managed by an application, such as Network Manager. NCIM holds entity data related to multiple domains.

Topology data

When the network is discovered, both *core* NCIM tables and *entity attribute* tables are updated with topology data. These tables include Layer 1, Layer 2, Layer 3, device structure, routing protocol, containment, and technology-specific information.

The NCIM tables are not case-sensitive.

The core and entity attribute tables are listed in the data dictionary.

Data modeled by NCIM

NCIM models different types of network data, including the following:

IP networks
NCIM models network devices and device connections within layer 2 and layer 3 of the OSI model.

Optical networks
NCIM models optical network devices within layer 1 of the OSI model.

Routing networks
NCIM models routing networks based on routing protocols such as MPLS, BGP, and OSPF.

VLANs
NCIM models Virtual Local Area Networks (VLANs).

Radio access networks
NCIM models a variety of radio access networks (RANs), including GSM, UMTS, and LTE.

Related concepts

Data dictionary
The NCIM topology database schema is made up of a set of relational database tables that represent the topology model.

Domains and entities

The NCIM topology database models network domains and entities within those domains.

Domains

A domain is a scoped set of entities that are discovered and managed by an application, such as Network Manager. NCIM holds entity data related to multiple domains.

For more information on domains, see the *IBM Tivoli Network Manager User Guide* and the *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*.

Entities

A Network Manager discovery returns many different types of entity. If you understand the entities that you might encounter, you can more easily restrict your queries to return only required information.

Basic information about discovered network resources is held within the `entityData` table. Resource-specific attribute data is held in product-specific extension tables that typically have a foreign key relationship with the core-model `entityData` table. Information on relationships between discovered network resources, such as containment and connectivity, is also held in tables, such as the `contains` and `connects` tables.

Records in the `entityData` table are at least related to a given instance and the `domainMgr`, `manager`, and `domainMembers` tables.

Discovered resources held in the `entityData` table can be any of the following types:

- *Physical and logical entities*, including devices and their physical and logical characteristics, such as slots, cards, ports, and interfaces, and the relationships between them.
- *Protocol end points* represent protocol or technology-specific information that is typically associated with an entity representing a port or interface resource.
- *Device collections*, including MPLS VPNs, global VLANs and subnets.
- *Hosted services*, including BGP and OSPF services hosted on a device.

Network Manager populates the database with information about discovered layer 1, layer 2 and layer 3 entities. To uniquely identify entities as they are discovered, the NCIM database uses a unique key, entityId. The entityId column appears in all database tables that reference entities.

For example, the entityId column appears in the core `entityData` table, as well as in the `physicalChassis`, `networkInterface`, and `physicalPowerSupply` tables.

The following table describes the discovery-related entities that are stored in the NCIM database.

Some entity types are defined in advance for future use.

Table 256. Network Manager entities

Entity type	Entity type name	Category	NCIM table	Description
0	Unknown	Element		
1	Chassis	Element	physicalChassis	Main node device.
2	Interface	Element	networkInterface	Interfaces with entityType 2 can be discovered and polled.
3	Logical Interface	Element	networkInterface	Interfaces with entityType 3 are inferred but are not directly accessible. Hot Standby Routing Protocol (HSRP) virtual IP interfaces are an example of logical interfaces.
4	Local VLAN	Element	localVlan	VLAN port on a main node device.
5	Module	Element	physicalCard	Card within a switch or router. The term *module* is used to avoid confusion with the term *card* which is used in layer 1 networks.
6	PSU	Element	physicalPower Supply	Power supply unit within a main node device.
7	Logical Collection	Collection		Examples of logical collections include MPLS VPNs, global VLANs and subnets. NCIM can also model OSPF areas.
8	Daughter Card	Element		The child of a network card.
9	Fan	Element	physicalFan	Fan component within a main node device.
10	Backplane	Element	physicalBackplane	Backplane component within a main node device. Backplanes usually contain slot entities.
11	Slot	Element	physicalSlot	Slot component within a main node device. Slots usually contain module entities.
12	Sensor	Element	physicalSensor	Sensor component within a main node device.
13	Virtual Router	Element	virtualRouter	Represents a instance of a virtual router within a chassis device.
14	CPU	Element	cpu	Represents Central Processing Units (CPUs).
15	Subnet	Collection	subnet	Logical collection that lists the IP address in a class A, B, or C subnet.
16	Global VLAN	Collection	globalVlan	Collection of VLAN entities across multiple chassis devices that combine to form a virtual network.
17	VPN	Collection	networkVpn	Logical collection of IP address collected within a VPN.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
18	HSRP Group	Collection	hsrpGroup	Represents an Hot Standby Routing Protocol (HSRP) group logical collection. The Cisco HRSP implements a virtual router with its own IP and MAC addresses. This virtual router forms an HSRP group that consists of a number of real interfaces, only one of which is active at any given time. The active interface forwards IP traffic that is sent to the virtual router, and the other interfaces in the group stand by ready to become active if the active interface fails.
19	Stack	Element		Collection of chassis devices as defined by the Entity MIB.
20	VRF	Element	vpnRouteForwarding	Represents a VPN routing and forwarding table.
21	OSPF Routing Domain	Collection	ospfRoutingDomain	Represents an OSPF routing domain.
22	OSPF Service	Service	ospfService	Represents an OSPF service running on a device.
23	OSPF Area	Collection	ospfArea	Represents an OSPF area.
24	VTP Domain	Collection	vtpDomain	Represents a VLAN trunking protocol domain.
25	Other	Element	physicalOther	Stores attributes of a component whose entity type the discovery was unable to determine. This occurs if the physical entity class is known, but does not match any of the supported values.
26	BGP Service	Service	bgpService	Represents a BGP service.
27	BGP AS	Collection	bgpAutonomousSystem	Represents a BGP autonomous system.
28	BGP Route	Attribute	bgpRouteAttribute	Represents a BGP route.
29	BGP Cluster	Collection	bgpCluster	Represents a BGP cluster.
30	BGP Network	Service	bgpNetwork	Represents a BGP network.
31	ISIS Service	Collection		Represents an ISIS service.
32	ISIS Level	Element		Represents the ISIS level.
33	OSPF Pseudo-Node	Element		Represents an OSPF pseudo-node.
34	ITNM Service	Collection	itnmService	The base type for other services such as ISIS Service.
35	MPLS TE Service	Service	mplsTEService	Represents a Multi Protocol Label Switching Traffic Engineered (MPLS TE) service

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
36	MPLS TE Tunnel	Element	mplsTETunnel	Represents an MPLS TE tunnel
37	MPLS TE Resource	Element	mplsTETunnel Resource	Represents an MPLS TE resource
38	MPLS LSP	Element	mplsLSP	Represents an MPLS Label Switch Path (LSP)
40	IP Connection	Element	ipConnection	Represents a connection using TCP/IP.
41	PIM Service	Service	pimService	Represents a Protocol Independent Multicast (PIM) service.
42	PIM Network	Collection	pimNetwork	Represents a PIM network.
43	IPMRoute Service	Service	ipMRouteService	Represents an IP Multicast Routing service.
44	IPMRoute Upstream	Element	ipMRouteUpstream	Stores the upstream (RPF) route statistics for each device or Multicast Distribution Tree (MDT).
45	IPMRoute Downstream	Element		Stores the downstream route statistics per device or MDT.
46	IPMRouteMdt	Collection	ipMRouteMdt	Stores the Collection entities representing the MDTs for each Multicast Source or Group.
47	IPMRouteSource	Element	ipMRouteSource	Represents Multicast Sources, as contained by the MDT.
48	IPMRouteGroup	Element	ipMRouteGroup	Represents Multicast Groups, as contained by the MDT.
49	IP Path	Collection	ipPath	Represents a network path between IP devices.
50	IP End point	Protocol Endpoint	ipEndPoint	Represents a logical IP end point that is implemented by a physical interface.
51	VLAN Trunk End point	Protocol Endpoint	vlanTrunkEndPoint	Represents a logical VLAN trunk end point that is implemented by a physical interface.
52	Frame Relay End point	Protocol Endpoint	frameRelay EndPoint	Represents a logical Frame Relay end point that is implemented by a physical interface.
53	OSPF End point	Protocol Endpoint	ospfEndPoint	Represents a logical OSPF end point that is implemented by a physical interface.
54	ATM End point	Protocol Endpoint	atmEndPoint	Represents a logical ATM end point that is implemented by a physical interface.
55	VPWS End point	Protocol Endpoint	vpwsEndPoint	Represents a logical VPWS end point that is implemented by a physical interface.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
56	BGP End Point	Protocol Endpoint	bgpEndPoint	Represents a logical BGP end point that is implemented by a physical interface.
57	ISIS End Point	Protocol Endpoint		Represents a logical ISIS end point that is implemented by a physical interface.
58	MPLS Tunnel End Point	Protocol Endpoint	mplsTETunnelEndPoint	Represents a logical MPLS tunnel end point that is implemented by a physical interface.
59	TCP/UDP End Point	Protocol Endpoint		Represents a logical TCP/UDP end point that is implemented by a physical interface.
60	PIM End Point	Protocol Endpoint	pimEndpoint	Represents the Protocol Independent Multicast (PIM) end points discovered in the network and their associated attributes.
61	IPMRoute End Point	Protocol Endpoint	ipMRouteEndPoint	Stores information on the IP Multicast Routing Protocol End Points.
62	IGMP End Point	Protocol Endpoint	igmpEndPoint	Stores information on the Internet Group Membership Protocol (IGMP) End Points.
63	Network Service Entity End Point	Protocol Endpoint	networkServiceEntityEndPoint	Helps model relationships related to the management of frame relay links.
67	LAG End Point	Protocol Endpoint	lagEndPoint	Represents a logical Link Aggregation Group (LAG) end point that is implemented by a physical interface.
68	Probe End point	Protocol Endpoint	probeEndPoint	Fix Pack 3 Represents the source or target end point of a probe operation, implemented by a physical interface.
70	Topology	Topology		Grouping of connections which belong to a topology.
71	Layer 1 Topology	Topology		Grouping of connections which belong to a Layer 1 topology.
72	Layer 2 Topology	Topology		Grouping of connections which belong to a Layer 2 topology.
73	Layer 3 Meshed Topology	Topology		Grouping of connections which belong to a Layer 3 meshed topology.
74	Converged Topology (Layer 1 - Layer 3)	Topology		Based on data available in NCIM, groups together connections at the lowest layer for which data is available.
75	MPLS TE Topology	Topology		Grouping of connections which belong to an MPLS TE topology.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
77	Pseudo Wire Topology	Topology		Grouping of connections which belong to a Pseudo Wire topology.
78	OSPF Topology	Topology		Represents an OSPF topology.
79	BGP Topology	Topology		Represents a BGP topology.
80	IP Path Topology	Topology	ipPath	Represents an IP path.
81	PIM Topology	Topology		Represents PIM topologies.
82	Local VLAN Topology	Topology		Represents local VLAN topologies.
83	IPMRoute Topology	Topology		Represents an IP Multicast Routing topology.
84	VPLS Pseudo Wire Topology	Topology		Respresents a VPLS Pseudo Wire Topology.
85	Virtualization Topology	Topology		Represents a virtualization topology.
86	Microwave Topology	Topology		Represents a microwave topology.
87	RAN Topology	Topology		Represents a radio access network topology.
90	LTE Control Plane	Topology		Represents the devices and connectivity that make up the LTE control plane.
91	LTE User Plane	Topology		Represents the devices and connectivity that make up the LTE user plane.
92	Probe Topology	Topology		Represents the probe source/target connectivity.
110	Generic Collection	Collection	genericCollection	A collection that is not of any other type.
111	Geographic Location	Element	geographicLocation	Represents a geographic location.
112	Geographic Region	Collection	geographicRegion	Represents a geographic region.
113	VLAN Ports	Collection	vlanCollection	Represents a collection of the ports on a given named VLAN or, if no name is provided, on a given VLAN identifier.
120	IGMP Service	Service	igmpService	Represents an Internet Group Management Protocol (IGMP) service.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
121	IGMP Groups	Collection	igmpGroup	Stores multicast group collections for which there are associated Internet Group Membership Protocol (IGMP) end points in the igmpEndPoint table.
122	VSI (Virtual Switch Instance)	Element	virtualSwitch Instance	Represents a virtual switch instance (VSI) configured on a Provider Edge (PE) device that is associated with a Virtual Private LAN Service (VPLS) Virtual Private Network (VPN) instance.
123	Data Center	Element		Represents a data center.
124	Virtual Cluster	Collection	virtualCluster	Represents a cluster of virtual machines.
125	Virtual Management Service	Service	virtualMgmtService	Represents a virtual management service.
126	Hypervisor	Element	hypervisor	Represents a hypervisor.
127	Port Group	Collection	portGroup	Represents a port group.
128	EMS System	Element	emsSystem	Represents an EMS system accessed by a collector.
130	RAN GSM Cell	Element	ranGSMCell	Represents a GSM cell.
131	RAN UTRAN Cell	Element	ranUtranCell	Represents a UTRAN cell.
132	RAN Sector	Element	ranSector	Represents a RAN sector.
133	RAN NodeB Local Cell	Element	ranNodeBLocalCell	Represents a NodeB Local Cell.
134	RAN Location Area	Collection	ranLocationArea	Represents a RAN Location Area.
135	RAN Routing Area	Collection	ranRoutingArea	Represents a RAN Routing Area.
136	RAN Packet Core	Collection		Represents RAN packet switch core entity.
137	RAN Circuit Core	Collection		Represents a RAN circuit switched core entity.
138	RAN Radio Core	Collection	ranRadioCore	Represents a RAN radio core entity.
139	RAN Transceiver	Collection	ranTransceiver	Represents a RAN transceiver.
150	LTE Sector	Element	eUtranSector	Represents a geographic area of radio coverage and is implemented and supported by physical radio equipment. An LTE sector implements one or more LTE cells.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
151	LTE Cell	Element	eUtranCell	Represents a geographical area of radio coverage and is implemented and supported by physical radio equipment, such as towers, amplifiers, and antennas.
152	MME Function	Element	mmeFunction	The Mobility Management Entity (MME) is the main signalling control element in the core network and is the key control node for enabling user equipment access to the core network. The role of the MME is implemented within a network hardware node and is modelled by NCIM using the mmeFunction entity type. Multiple mmeFunction instances can be implemented within a single network hardware node.
153	Tracking Area	Collection	trackingArea	Represents an LTE tracking area.
154	SGW Function	Element	sgwFunction	The Serving Gateway (SGW) resides in the user plane where it forwards and routes packets to and from the eNodeB and packet data network gateway (PGW). The role of the SGW is implemented within a network hardware node and is modelled by NCIM using the sgwFunction entity type. Multiple sgwFunction instances can be implemented within a single network hardware node.
155	PGW Function	Element	pgwFunction	The Packet Data Network Gateway (PGW) provides user plane connectivity to packet data networks. The role of the PGW is implemented within a network hardware node and is modelled by NCIM using the pgwFunction entity type. Multiple pgwFunction instances can be implemented within a single network hardware node.
156	ENB Function	Element	enbFunction	The eNodeB device manages the radio air interface communication with users of the LTE network. Each eNodeB device controls one or more cells, which are geographic areas of radio coverage. The role of the eNodeB is implemented within a network hardware node and is modelled by NCIM using the enbFunction entity type. Multiple enbFunction instances may be implemented within a single network hardware node.
157	LTE Pool	Collection	ltePool	Generic modelling mechanism for groups of pooled LTE entities, and currently used to model MME pools, PGW pools, and SGW pools. As an example, in order to model an MME pool, the relationship between the ltePool entity and associated mmeFunction entities is modelled using the collects table.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
158	PLMN	Element	plmn	Models a Public Land Mobile Network (PLMN). A PLMN is a network that provides land mobile telecommunications services to the public. Each operator providing mobile services has its own PLMN.
159	HSS Function	Element	hssFunction	Models the LTE Home Subscriber Service (HSS). The HSS manages subscriber identities, service profiles, authentication, authorization, and quality of service (QoS), and acts as the master repository for subscriber profiles, device profiles and state information.
160	PCRF Function	Element	pcrfFunction	Models the LTE Policy and Charging Rules Function (PCRF). The PCRF manages the policy and charging for uplink and downlink service flows and the permitted EPS bearer QoS.
161	EIR Function	Element	eirFunction	Models the LTE Equipment Identity Register (EIR). The EIR keeps track of mobile devices which should either be banned from using the network or monitored. When a mobile phone is stolen its identity it is added to the EIR blacklist and the result is that this phone will never be able to attach to the network for service. Usually each network has its own EIR which is often combined with the HSS node. It is possible for multiple operators to share a common EIR which enables the blacklisted information to be more easily and widely available.
163	LTE Control Plane	Collection	controlPlane ViewCollection	Supports the dynamic collection views under **LTE Network Topology** > **Control Plane by Tracking Area** in the Network Views. Each instance of this entity type collects the eNodeBs in the corresponding tracking area, together with the devices that these eNodeBs are connected to on the control plane.
164	LTE User Plane	Collection	userPlane ViewCollection	Supports the dynamic collection views under **LTE Network Topology** > **User Plane by Tracking Area** in the Network Views. Each instance of this entity type collects the eNodeBs in the corresponding tracking area, together with the devices that these eNodeBs are connected to on the user plane.
170	Aggregated Link	Collection	aggregatedLink	Represents a network link between Link Aggregation Groups (LAGs)
171	Link Aggregation Group	Element	aggregationGroup	Represents a Link Aggregation Group (LAG).
190	Probe Service	Service	probeService	Fix Pack 3 Represents the service that provides probes on a device.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
191	Probe	Collection	probe	Fix Pack 3 Represents configured network probes and their attributes.
192	Probe Collection	Collection	probeCollection	Fix Pack 3 Provides a collection facility for probes or probe collections.
200	LTE S1-U	Topology	entityData	Topology type for LTE S1-U connectivity.
201	LTE S5	Topology	entityData	Topology type for LTE S5 connectivity.
202	LTE S8	Topology	entityData	Topology type for LTE S8 connectivity.
203	LTE S1-MME	Topology	entityData	Topology type for LTE S1-MME connectivity.
204	LTE S10	Topology	entityData	Topology type for LTE S10 connectivity.
205	LTE S11	Topology	entityData	Topology type for LTE S11 connectivity.
206	LTE SGi	Topology	entityData	Topology type for LTE SGi connectivity.
207	LTE Gx	Topology	entityData	Topology type for LTE Gx connectivity.
208	LTE S3	Topology	entityData	Topology type for LTE S3 connectivity.
209	LTE S4	Topology	entityData	Topology type for LTE S4 connectivity.
210	LTE S6a	Topology	entityData	Topology type for LTE S6a connectivity.
211	LTE S13	Topology	entityData	Topology type for LTE S13 connectivity.
212	LTE X2	Topology	entityData	Topology type for LTE X2 connectivity.
250	NR Sector	Element	eUtranSector	5G New Radio Sector Entity.
251	NR Cell DU	Element	NRCellDU	Represents a geographical area of radio coverage and is implemented and supported by physical radio equipment for 5G LTE, such as towers, amplifiers, and antennas.
252	NR Cell CU	Element	NRCellCU	Represents a geographical area of radio coverage and is implemented and supported by physical radio equipment for 5G LTE, such as towers, amplifiers, and antennas.

Table 256. Network Manager entities (continued)

Entity type	Entity type name	Category	NCIM table	Description
253	GNB DU Function	Element	gnbFunction	The gNodeB device manages the radio air interface communication with users of the 5G network. Each gNodeB device controls one or more cells, which are geographic areas of radio coverage. The role of the gNodeB is implemented within a network hardware node and is modelled by NCIM using the gnbFunction entity type. Multiple gnbFunction instances may be implemented within a single network hardware node.
254	GNB CUCP Function	Element	gnbFunction	The gNodeB device manages the radio air interface communication with users of the 5G network. Each gNodeB device controls one or more cells, which are geographic areas of radio coverage. The role of the gNodeB is implemented within a network hardware node and is modelled by NCIM using the gnbFunction entity type. Multiple gnbFunction instances may be implemented within a single network hardware node.
255	GNB CUUP Function	Element	gnbFunction	The gNodeB device manages the radio air interface communication with users of the 5G network. Each gNodeB device controls one or more cells, which are geographic areas of radio coverage. The role of the gNodeB is implemented within a network hardware node and is modelled by NCIM using the gnbFunction entity type. Multiple gnbFunction instances may be implemented within a single network hardware node.

Related reference

entityType

The entityType table provides a comprehensive list of every entity type in NCIM. It belongs to the category *entities*.

entityData table and entity view

Information on entities is held in the entityData table in Network Manager versions 3.9 and later. This table replaces the entity table used in earlier versions. To ensure backward compatibility an entity view has been created to hold the same data as the entity table from earlier versions.

The difference between the entityData table and the earlier entity table is that entities in the entityData can be members of more than one domain. In versions 3.8 and earlier, an entity in the entity table could only be a member of a single domain.

In order to facilitate this, the domainMgrId field that was in the earlier entity tables does not appear in the entityData table. Instead, in versions 3.9 and later a new domainMembers table maps entityId values from the entityData table to the domainMgrId values from the domainMgr table. This enables a single entity to be a member of multiple domains.

In versions 3.9 and later the entity view is created by joining the two tables, entityData and domainMembers.

Related reference

domainMembers
The `domainMembers` table stores information on membership of entities within domains. This table belongs to the category *domains*.

domainMgr
The domainMgr table stores data on network domains. This table belongs to the category *domains*.

entity
The `entity` view joins data from the `entityData` and `domainMembers` tables and is equivalent to the `entity` table that existed in Network Manager versions 3.8 and earlier.The `entity` view stores data on entities and includes the domainMgrId, which the domain in which the entity is located.

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

Protocol-specific data

Device entities, usually interfaces, can be associated with protocol-specific data. One example is the association of a device interface with the IP addressing data. Ports and interfaces can also be associated with other data, including ATM, BGP and OSPF protocol endpoints.

NCIM associates protocol-specific information with entities such as interfaces, using protocol endpoint tables. Examples of protocol endpoint tables are the `atmEndPoint` and `ipEndPoint` tables.

Technology-specific data

NCIM models a range of different network technologies, including IP, VLANs, and MPLS VLANs.

Related reference

ipEndPoint
The ipEndPoint table represents an IP end point and includes relevant data. The endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

Relationships

The NCIM topology database models relationships between entities.

Connectivity data

Connectivity data defines how entities are connected in the network. It includes connections between different devices, and VLAN-related connections within the same device. Connectivity information is stored in the `topologyLinks`, `networkPipe`, and `pipeComposition` tables.

Bidirectional connections are only entered into the database once, either from the "A" end to the "Z" end or from the "Z" end to the "A" end. Therefore, SQL queries that extract connectivity data must check for the connection in either direction.

Representation of connectivity at different layers of the topology

The NCIM database represents the connectivity of entities in different independent layers. Therefore representation of connectivity at layer 2 is represented independently of the connectivity at layer 3. Each connection is associated with a topology entity.

Representation of connectivity within sub-topologies

The NCIM database represents complex connectivity scenarios. For example, within the MPLS VPN realm, NCIM can model the layer 3 connection between a provider-edge (PE) router and multiple customer-edge (CE) routers. Connectivity between multiple devices that form a mini-topology is defined in the topologyLinks table.

Related reference

Find devices connected to a named device
This query identifies all main node devices connected to a single specified main node device.

Containment data

Containment data defines logical and physical containment within your network. A *containment model* is generated at the end of the discovery process when the network topology is created. This model reflects the real-world topology of the network that is being modelled, in a physical, logical or business-oriented sense.

Overview of containment

Containment is a key principle of the network model. Containers are objects that can "hold" both *elements* and other containers. Elements and containers can represent logical or physical entities. You can put any object within a container and even mix different objects within the same container.

An example of *physical containment* is a chassis containing network interface cards; the network interface cards can themselves contain individual ports. An example of *logical containment* is a set of ports or interfaces being contained by a particular VLAN. Network Manager also uses VLAN objects to model containment. VLAN objects are created by the stitchers. They contain all the interfaces that exist on each VLAN.

Use of the containment model

When generated, the default containment model represents both physical and logical containment:

- The physical containment model enables you to perform device management down to the port level.
- The logical containment model shows how objects are contained within logical containers that do not necessarily exist in the physical sense. One example is a VLAN container, which is a logical grouping of devices, cards, and ports that are not necessarily physically connected together or in the same location. The default logical containment model is based on VLAN containment.

VLAN naming

Network Manager uses different naming conventions. One approach is to identify the entity name, card and port numbers of specific ports, in the format `EntityName [ card [ port ] ]`.

For example, port 12 on card 1 of chassis A is identified as `A[ 1 [ 12 ] ]`.

By using stitchers, VLAN names can also be modified to reflect the business context in which a given VLAN is used.

The naming used also depends on configuration of the product. This means that interface naming might be used; for example, `Se4/0`.

VLAN trunking

When traffic from different VLANs is passed along a single trunk link between switches, this is represented in the Network Manager containment model.

The following figure shows how the containment model represents this traffic.

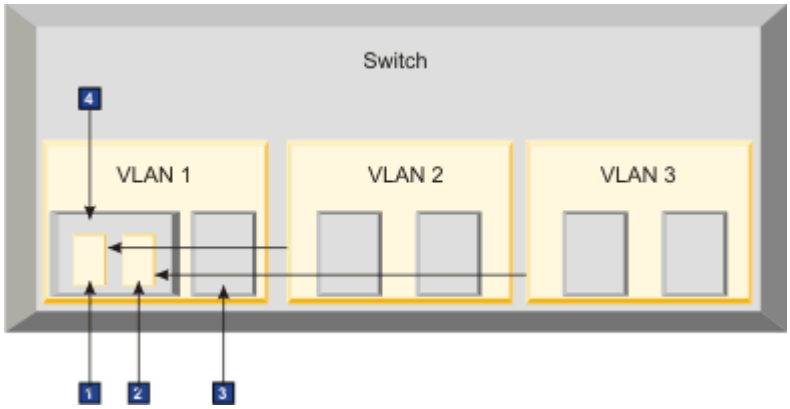


Figure 3. Logical sub-interfaces contained within a trunk port

If a port is being used for a trunk link, it contains logical sub-interfaces. The following information describes the properties of the ports and sub-interfaces shown in Figure 3 on page 491:

1 A sub-interface connecting VLAN 2 on the switch to VLAN 2 on another switch. Sub-interfaces are contained by trunk ports.

2 A sub-interface connecting VLAN 3 on the switch to VLAN 3 on another switch. Sub-interfaces have no upward connections to their containing trunk port.

3 A normal port.

4 A port containing sub-interfaces.

Customization of the containment model

The containment model can be customized. This customization is an advanced feature of the discovery process. To generate a custom containment model, you must either modify the existing stitchers, or write a new stitcher and configure the existing stitchers to run the new stitcher during the creation of the network topology.

Two example stitchers, `ExampleContainment1.stch` and `ExampleContainment2.stch` are supplied to help you modify the containment model. These stitchers can be executed by removing the comments before the `ExecuteStitcher();` statements in the `BuildContainment.stch` stitcher.

These stitchers are stored in the following directory: `$NCHOME/precision/disco/stitchers/`.

For a syntax definition of the stitcher language, see the *IBM Tivoli Network Manager Reference*.

Dependencies

When one entity in a system cannot meaningfully function without another entity it is said to be *dependent*. Dependencies can be defined by the containment model. A container can be dependent upon the objects it contains.

Network Manager applications take dependencies into account. The root-cause analysis (RCA) engine (a plug-in to the Event Gateway, ncp_g_event), for example, can consider dependencies when performing root cause analysis of network faults.

Collection data

Collection data defines logical collections. Collections are defined in the `collects` table. Examples of logical collections defined within NCIM include MPLS VPNs, global VLANs, and subnets.

NCIM can also model OSPF areas. Each row in the `collects` table holds a pair of entity identifiers: the collecting entity, for example the VPN, and one of the entities within that collecting entity.

Related reference

Find all devices in a given VPN

This query identifies all of the VPNs listed in the database. For each VPN the query provides the name of that VPN and the list of IP addresses collected within that subnet. The IP address collected within a VPN might refer to main nodes or interfaces; typically they refer to interfaces.

collects
The `collects` table stores data on collections of entities, such as subnets and MPLS VPNs. This table belongs to the category *collections*.

Hosted services

A hosted service is a service or application running on a specific device. For example, a device can host BGP or OSPF services. NCIM can also model the fact that a software application, is running on a workstation.

Related reference

Find all chassis devices hosting OSPF services
This query identifies all devices that are hosting OSPF services. These devices are serving as routers within an autonomous system (AS). Each device identified has an IP address and a separate OSPF router IP address.

NCIM cache files

Topology updates are held in a set of files called the NCIM cache files.

There is one cache file for each type of update message and the name of each cache file reflects the content. The format of each file matches the data in the ncimCache database tables. The cache files are as follows:

- `NCHOME/var/precision/Store.Cache.ncimCache.collects.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.connectActions.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.connects.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.contains.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.dependency.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.domainMembers.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.entityActions.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.entityData.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.hostedService.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.lingerTime.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.managedStatus.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.networkPipe.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.pipeComposition.`*`DOMAIN`*
- `NCHOME/var/precision/Store.Cache.ncimCache.protocolEndPoint.`*`DOMAIN`*

Where *DOMAIN* is the current domain.

Related reference

ncimCache database
This database stores topology updates from DNCIM.

SQL files for the NCIM schema

The NCIM database schema is contained within several SQL files. The following is a list of SQL files that contain the schema.

The files are as follows:

DB2

```
$NCHOME/precision/scripts/sql/db2/createPrecisionMgmtTables.sql
$NCHOME/precision/scripts/sql/db2/createNetCoolCoreDb.sql
```

Oracle

```
$NCHOME/precision/scripts/sql/oracle/createPrecisionMgmtTables.sql
$NCHOME/precision/scripts/sql/oracle/createNetCoolCoreDb.sql
```

The schema files below are common to all database products.

- `$NCHOME/precision/scripts/sql/data/populateMappings.sql`
- `$NCHOME/precision/scripts/sql/data/populateEnumerations.sql`
- `$NCHOME/precision/scripts/sql/data/populateDeviceFunction.sql`
- `$NCHOME/precision/scripts/sql/data/populateDefaults.sql`

The database schema specific to Network Manager is contained in the `createPrecisionIPDb.sql` file.

The directory location of the Network Manager database schema is as follows:

DB2

```
$NCHOME/precision/scripts/sql/db2/createPrecisionIPDb.sql
```

Oracle

```
$NCHOME/precision/scripts/sql/oracle/createPrecisionIPDb.sql
```

To better understand how to formulate queries for purposes of correlating, analyzing, or reporting data, you can view these files, but do not attempt to modify them.

Chapter 21. Topology database queries

Use these sample SQL queries, which are based on real-world queries, as an example of the kind of data that can be extracted, and as a basis for constructing further queries.

About this task

Different databases can require differently formed queries. Refer to your database documentation for the required format of queries. This information assumes that you are familiar with SQL syntax. For more information about SQL, refer to an appropriate SQL tutorial or reference text.

Note: Earlier versions of this documentation contained a section on Extending the NCIM topology database. This section has now been replaced by the new topology enrichment features. For more information see the Enriching the topology chapter within *IBM Tivoli Network Manager IP Edition Administration Guide*.

Logging in to NCIM

Log in to NCIM to run an SQL query that retrieves topology data.

About this task

To log in to NCIM using **ncp_oql** you must provide a valid NCIM user name and password. The default user name for the NCIM database user is ncim. The default password is ncim.

To log in to NCIM enter the following command:

```
ncp_oql -domain DOMAIN -service ncim -username USERNAME -password PASSWORD
```

Use the tabular display format capabilities of the `ncp_oql` command. The -tabular option is useful when retrieving only a small number of columns. For more information on the `ncp_oql` command, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Formatting used in the SQL queries

The SQL queries are formatted for readability.

The following formatting is used:

- SQL keywords, such as SELECT and INNER JOIN, are presented in uppercase.
- Code is spaced to aid scanning.
- Each piece of data extracted by a SELECT statement is presented on a separate line.
- Capitalization is used within table and field names. For example, in the field name mainNodeEntityId the *M*, *E*, and *I* are capitalized.

  Note: Capitalization of table and field names is not required in the SQL queries that you submit to NCIM.

Techniques used in the SQL queries

The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

Related reference

Find devices connected to a named device

This query identifies all main node devices connected to a single specified main node device.

Choice of driving table

One of the most important design decisions when creating a query is the choice of driving table. The choice of *driving table* is particularly important for ensuring the efficiency of queries.

The driving table is the table from which rows of data are first selected. Data is then added from other tables by joining these tables, initially to the driving table. Therefore, choose the driving table so that a minimum of rows are initially selected. This ensures that the query is as efficient as possible. In many of the sample queries, the driving table is the domainMgr table, as there are generally very few rows in this table. This is in contrast to the `entityData` table, which generally holds tens or hundreds of thousands of rows.

Aliasing

Aliasing is the use of a temporary name for a column, sub-query or table within a query.

Common reasons for using aliasing include:

- Brevity: For example, use *e* to refer to the `entityData` table.
- Distinguishing between table data in a meaningful way: For example, use *eComponent* to refer to the `entityData` table when extracting component data from this table. Use *eMainNode* to refer to the `entityData` table when extracting main node data from the table.

Aliasing can also be applied to columns, functions, and subqueries. For example, aliasing can be used to rename a results column.

Table joins

Use table joins to combine records from one or more tables. Two types of table join are used, INNER JOIN and OUTER JOIN.

OUTER JOIN

An OUTER JOIN table join preserves all the rows in one or both tables, even when they do not have corresponding rows in the other tables being queried. An example of when an OUTER JOIN table join is useful is if you want to retrieve all interface and IP addressing data where applicable, bearing in mind that some interfaces may not have IP addresses. Commonly used outer joins include:

LEFT OUTER JOIN

Retains all records from the left table even if the join predicate does not find any matching record in the right table.

RIGHT OUTER JOIN

Retains all records from the right table even if the join predicate does not find any matching record in the left table.

INNER JOIN

An INNER JOIN table join between tables combines the records from one or more tables based on a given join predicate to produce a record set that incorporates rows and columns from each table included in the join. Typically, a common attribute, such as the NCIM entityId, is used to retrieve sets of associated records. For example, an inner join could be used to retrieve all of the records that contain other resources by joining the entity.entityId and contains.containingEntityId attributes.

Use of specific fields and tables in queries

You can write more efficient SQL queries by making careful use of certain strategic fields and tables.

mainNodeEntityId field

The mainNodeEntityId field in the `entityData` table specifies the main node of the entity. This field provides a shortcut to the main node for a particular entity, avoiding the need to traverse the entire containment tree.

The mainNodeEntityId field is relevant only for entities that are wholly contained within a single main node device. It therefore has a non-NULL value only for entities that are related to a single main node device, such as:

- The main node itself
- Physical and logical device components, such as interfaces, modules, PSUs, sensors, backplanes, and fans
- Logical interfaces entities on the main node, such as IP endpoints and VLAN trunk endpoints
- Local VLANs, which are local VLAN entities contained within a single main node device. The interfaces contained by this VLAN are constrained to only those interfaces contained within the main node device.

Entities that are related to multiple main node devices, such as VPNs and global VLANs, have a NULL value in the mainNodeEntityId field.

To retrieve only the entities that are wholly contained within a single main node device, use an INNER JOIN statement on the `entityData` table. This statement ensures that only entities that have a non-NULL value in the mainNodeEntityId field are retrieved.

entityType field

The entityType field can be used in SQL queries to limit the type of component data that is retrieved.

For example, if you specify the entity type 2, which corresponds to interfaces, in an SQL query, only component data of the type "interface" is retrieved. The entity type of each entity is specified in the entityType field of the `entityData` table.

Protocol endpoint tables

The protocolEndPoint and ipEndPoint tables can be used in SQL queries to identify the IP addresses that are implemented by the device interfaces.

protocolEndPoint
: This table associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The most common example of the contents of the protocolEndPoint table is a row of data that associates a device interface with the IP addressing data associated with that interface. The protocolEndPoint table refers to protocol-specific information, such as IP addressing data, using an entity ID.

ipEndPoint
: This table contains the IP addressing data.

Protocols other than IP have their own protocol endpoint tables, for example:

- atmEndPoint table for ATM
- bgpEndPoint for Border Gateway Protocol (BGP)
- frameRelayEndPoint for Frame Relay
- igmpEndPoint for Internet Group Multicast Protocol (IGMP)
- ipmRouteEndPoint for IP Multicast routes
- mplsTeTunnelEndPoint for Traffic Engineering tunnels
- OSPFEndPoint table for OSPF

- pimEndPoint for Protocol Independent Multicast
- portEndPoint for ports
- vlanTrunkEndPoint table for VLAN trunks
- vpwsEndPoint for Virtual Private Wire Services

Related reference

protocolEndPoint
The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

Queries for domain information

These queries retrieve information relevant to an entire domain or multiple domains. A domain is a scoped set of entities discovered and managed by an application. Sample queries extract information on the number of devices in a domain, names of devices in a domain, and so on.

Tip: A single SQL query on the NCIM database can extract data from multiple domains, whereas queries on the MODEL topology database, which can extract information from only a single domain at a time.

List all main nodes in a domain

This query provides a list of all main nodes in the database for a specified domain, or for all domains. The query provides the entity ID of the main node together with the entity name.

Entity ID
The unique primary key of the entity within the `entityData` table. This is an integer value assigned to the entity by the database. Entities are not only main nodes. Entities include any device component present in the database, and other items recorded in the database, such as collections of logical or physical network elements, for example, VPNs and VLANs.

Entity name
A string value used to refer to the entity. If the entity is a device, then the entity name might be the IP address of the device or the device name.

Note: Entity names are unique only within a given domain.

Example

```
SELECT     e.entityId Entity_ID,
           e.entityName Device_Name
FROM       domainMgr d
INNER JOIN entity e ON e.domainMgrId = d.domainMgrId
WHERE      d.domainName = 'NCOMS'
AND        e.entityType = 1
```

Description

The table below describes this query.

Table 257. Description of the query

Line numbers	Description
1-2	Specify the data to show in the results as follows: • The entity identifier of a main node device, represented by e.entityId. • The entity name of the device, represented by e.entityName.
3	Specify the domainMgr table as the driving table for this query.

Table 257. Description of the query (continued)	
Line numbers	**Description**
4	Retrieve all entities in each domain by joining the entity view. The INNER JOIN ensures that only entities that are associated to a domain (that is, with a valid domainMgrId field) are retrieved.
5	Restrict the results to entities within the NCOMS domain. **Tip:** To list all main node devices across all domains, omit this line from the query.
6	Restrict the entities to main nodes. Entities that have an entityType of 1 are main nodes.

Results

The table below shows a portion of the results of this query.

Table 258. Results of the query	
Entity ID	**Device name**
1	192.168.15.23
3	192.168.15.7
5	192.168.15.21
72	VE002.example.net
74	172.20.4.20
77	172.20.4.16
83	172.50.0.2
98	VE003.example.net
109	10.1.254.7
143	10.1.254.30
269	10.1.1.9

Related reference

Choice of driving table

One of the most important design decisions when creating a query is the choice of driving table. The choice of *driving table* is particularly important for ensuring the efficiency of queries.

Count the number of entities in a domain

This query counts the total number of entities in a domain. Entities include any device component present in the database, as well as other items recorded in the database such as collections of logical or physical

network elements, for example, VPNs and VLANs. This query returns a number indicating the number of entities in the domain.

Example

```
1] SELECT     COUNT(*)
2] FROM       domainMgr d
3] INNER JOIN entity e ON e.domainMgrId = d.domainMgrId
4] WHERE      d.domainName = 'NCOMS'
```

Description

The table below describes this query:

Table 259. Description of the query

Line number(s)	Description
1	Specify that we wish to count the number of rows returned by the query. Each entity returned by the query generates a row of results; therefore the number of rows returned corresponds to the number of entities in the domain.
2	Specify the `domainMgr` table as the driving table for this query.
3-4	Retrieve all entities in the `NCOMS` domain, by joining the `entity` view. Restrict the domain to `NCOMS` by means of the `WHERE` statement.

Customizing the query

It is possible to customize this query to retrieve only entities of a specific type. You can find a complete listing of entity types by viewing the contents of the `entityType` table. To count the number of interfaces only in the domain, add the following line to the query:

```
AND        e.entityType = 2
```

In this line `e.entityType` is the `entityType` field within the `entity` view. The `entity` view is referred to using the alias `e`.

```
1] SELECT     COUNT(*)
2] FROM       domainMgr d
3] INNER JOIN entity e ON e.domainMgrId = d.domainMgrId
4] WHERE      d.domainName = 'NCOMS'
5] AND        e.entityType = 2
```

Description

The table below describes this customized query:

Table 260. Description of the customized query

Line number(s)	Description
1	Specify that you want to count the number of rows returned by the query. Each entity returned by the query generates a row of results; therefore the number of rows returned corresponds to the number of entities in the domain.
2	Specify the `domainMgr` table as the driving table for this query.

Table 260. Description of the customized query (continued)	
Line number(s)	**Description**
3	Retrieve all entities in the NCOMS domain, by joining the `entity` view.
4	Restrict the results to entities within the NCOMS domain.
5	Restrict the entities returned by the query to interfaces. Entities with an `entityType` of 2 are interfaces.

Related reference

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

Choice of driving table
One of the most important design decisions when creating a query is the choice of driving table. The choice of *driving table* is particularly important for ensuring the efficiency of queries.

Queries for main node information

These sample queries retrieve data on main node devices.

List all devices with class name and system object identifier

This query retrieves all main node devices across all domains and, for each device, provides the class name of the device and the type of device.

Class name
The manufacturer and product family of the device. For example, CiscoCat35xx is the Cisco Catalyst 3500 product family.

Type of device
The model number of the device within product family of the manufacturer. For example, catalyst3524XL is the Cisco Catalyst 3524XL Gigabit Ethernet switch. The query determines the type of device by extracting the system object identifier (sysObjectId) value for the device. The sysObjectId field is held in the `chassis` table, which is one of the tables joined as part of the query. The system object identifier is a MIB value that provides the vendor's authoritative identification of the network management subsystem contained in the entity and serves as easy and unambiguous means for determining the type of device.

You can convert the system object identifier (for example, (1.3.6.1.4.1.9.1.248) into human-readable text (for example, catalyst3524XL), by using an OUTER JOIN statement to join the `mappings` table to the query. Within the `mappings` table, the sysObjectId mapping group lists system object identifier strings and their corresponding human-readable string values. The `mappings` table provides string-to-string mappings, unlike the `enumerations` table, which provides integer-to-string mappings.

If no entry exists in the `mappings` table for a specific system object identifier, the query returns a NULL value for the device type. Use of an OUTER JOIN statement enables you to perform this conversion without losing any rows of main node data. If no string exists for any particular system object identifier, the OUTER JOIN statement ensures that you nevertheless do not lose the row of data for the main node device associated with that system object identifier.

Example

```
1] SELECT     e.entityName Entity_Name,
2]            ec.className Class_Name,
3]            s.sysObjectId System_Object_ID,
```

```
4]            m.mappingValue Device_Type
5]  FROM      entityData e
6]  INNER JOIN physicalChassis c ON c.entityId = e.entityId
7]  INNER JOIN snmpSystem s ON s.entityId = e.entityId
8]  INNER JOIN classMembers cm ON cm.entityId = e.entityId
9]  INNER JOIN entityClass ec ON ec.classId = cm.classId
10]  LEFT OUTER JOIN mappings m ON m.mappingGroup = 'sysObjectId'
11]                 AND m.mappingKey = s.sysObjectId
12] ORDER BY   ec.className, s.sysObjectId
```

The following table describes this query:

Table 261. Description of the query

Line numbers	Description
1-4	Specify the data to show in the results, as follows: • The entity name of a device, represented by e.entityName. • The class name of the device, indicating the manufacturer and product family. This is represented by ec.className, where ec is the alias used to refer to the entityClass table. • The system object identifier for this device, represented by s.sysObjectId. (SNMP devices only) • The device type, based on a lookup of the system object identifier in the `mappings` table. This is represented by m.mappingValue.
5	Use the `entityData` table as the driving table for this query.
6	Limit the results of the query to main node devices, by joining the `physicalChassis` table to the `entityData` table. There is now a line of data for each main node device. Use an INNER JOIN statement to ensure that only entities that are main node devices are retrieved.
7	For SNMP devices determine the System Object ID.
8	Determine the class to which the device belongs. This is a two-step process. The first step, shown in this line, is to use an INNER JOIN statement to the classMembers table to retrieve the classId value for the class to which the device belongs.
9	Use the classId retrieved in line 7 as a lookup to determine the name of the class to which the device belongs. Do this by performing an INNER JOIN statement with the entityClass table. The entityClass table holds class details, including class names, and the name of the superclass, the containing class in the class hierarchy.
10-11	Look up the system object identifier in the mappings table in order to obtain a human-readable string for the device type. Do this by performing a join on the mappings table. Use an OUTER JOIN statement to enable you to perform this join without losing any rows of main node data. If no string exists for any particular system object identifier, the OUTER JOIN statement ensures that you nevertheless do not lose the row of data for the main node device associated with that system object identifier.
12	Order the query results for maximum readability. In order to do this list the devices first by manufacturer and product family (classname) and then by model (system object identifier).

Results

The table below shows a portion of the results of the query.

Table 262. Results of the query

Entity name	Class name	System object ID	Device type
192.168.15.23	3ComSuperStack	1.3.6.1.4.1.43.10.27. 4.1.2.2	3Com SuperStack II
192.168.15.7	3ComSuperStack	1.3.6.1.4.1.43.10.27. 4.1.2.2	3Com SuperStack II
172.20.4.16	Cisco26xx	1.3.6.1.4.1.9.1.185	cisco2610
10.1.1.8	Cisco26xx	1.3.6.1.4.1.9.1.186	cisco2611
10.1.1.9	Cisco26xx	1.3.6.1.4.1.9.1.209	cisco2621
172.20.4.15	Cisco36xx	1.3.6.1.4.1.9.1.122	cisco3620
10.1.254.1	Cisco72xx	1.3.6.1.4.1.9.1.222	cisco7206VXR
172.18.1.151	CiscoCat35xx	1.3.6.1.4.1.9.1.247	catalyst3512XL
172.18.1.203	CiscoCat35xx	1.3.6.1.4.1.9.1.248	catalyst3524XL
172.20.1.41	HuaweiARxx	1.3.6.1.4.1.2011.1.1.1. 12809	NULL
10.1.1.5	MarconiASX	1.3.6.1.4.1.326.2.2.5	NULL
192.168.32.13	Sun	1.3.6.1.4.1.42	NULL
192.168.34. 199	Sun	1.3.6.1.4.1.42.2.1.1	SunMicrosystemsServers
192.168.15.4	Windows	1.3.6.1.4.1.311.1.1.3.1.2	MicrosoftWindowsServer

List all IP addresses on all main node devices

This query retrieves all IP addresses on all main node devices. For each IP address, the query lists the entity that implements the IP address. This entity is usually an interface, but under certain conditions the IP address might be implemented by the main node itself.

This query lists the IP addresses implemented by each interface identified on a main node or by the main node itself. If an interface does not implement an IP address, that interface is not returned by this query.

Note: IP end points might be present on interfaces and on any of the following main nodes:

- Main nodes with no SNMP access
- Inferred chassis
- NAT-translated chassis

Example

```
SELECT     e.entityId Implementing_Entity_ID,
           eMainNode.entityName Main_Node_Name,
           e.entityName Implementing_Entity_Name,
           ip.address IP_Address
FROM       entityData e
INNER JOIN entityData eMainNode ON eMainNode.entityId =
                 e.mainNodeEntityId
INNER JOIN protocolEndPoint p ON p.implementingEntityId = e.entityId
INNER JOIN ipEndPoint ip ON ip.entityId = p.endPointEntityId
ORDER BY   e.entityId
```

Description

The table below describes this query.

Table 263. Description of the query

Line numbers	Description
1-4	Specify the data to show in the results, as follows: • The unique entity ID of the interface within the topology database, represented by `e.entityId` • The name of a main node device, represented by e`MainNode.entityName` • The name of the interface, represented by e`.entityName` • An IP address implemented by this interface, represented by `ip.address`
5	Use the `entityData` table as the driving table for this query. Use the alias e for the entityData table.
6-7	Identify the containing main node device for each of the entities retrieved in the preceding line. Do this by joining the `entityData` table to itself using the `mainNodeEntityId` field.
8-9	Identify the IP addresses implemented by each of the entities identified in line 5 of the query. Do this by performing an `INNER JOIN` statement on the `protocolEndPoint` table to extract the entity ID for any protocol-specific information associated with the entities identified in line 5. Then perform a second `INNER JOIN` statement on the `ipEndPoint` table to limit the protocol-specific information returned by the query to IP information.
10	To facilitate readability of the results, order first by the unique entity ID of the interface.

Results

The table below shows the results of this query.

Table 264. Results of the query

Implementing Entity ID	Main Node Name	Implementing Entity Name	IP Address
270	172.20.4.11	172.20.4.11[0[5]]	172.50.0.2
338	172.18.1.196	172.18.1.196[0[2]]	172.50.0.3
366	172.18.1.54	172.18.1.54[0[2]]	172.50.0.4
370	172.18.1.54	172.18.1.54[0[1]]	172.50.0.5
373	172.20.4.13	172.20.4.13[0[1]]	172.50.0.6
377	172.20.4.20	172.20.4.20[0[1]]	172.50.0.7

Table 264. Results of the query (continued)			
Implementing Entity ID	**Main Node Name**	**Implementing Entity Name**	**IP Address**
417	192.168.139.7	192.168.139.7[0 [5]]	172.20.11.1
417	192.168.139.7	192.168.139.7[0 [5]]	172.20.1.2

Queries for containment information

These queries retrieve data on logical and physical containment within your network.

The containment model can reflect the real world topology of the network that is being modelled, in a physical, logical or business-oriented sense. Logical containment includes the definition of local VLAN objects and VLAN trunks within main nodes.

The following sample queries extract containment information.

List all components on a device

This query retrieves all components on a named device, and lists each component. The query lists the components by entity ID and displays the name of the component. All components are displayed, regardless of their type.

You can run this query in the following ways, which specify the main node device differently:

- Using the device name (entityName) and the name of the domain in which the device is located (domainName). The device name might be an IP address or a textual name and should be unique within a given domain. This query is shown below.
- You can also write this query using the entityId for the device within the entityData table. This field contains an integer value unique across all domains.

Note: The SQL query in this section uses meaningful table aliases, such as eComponent and eMainNode . This makes the query more readable by enabling you to distinguish between different types of data from the same table.

Example

```
1] SELECT     eComponent.entityId Component_Entity_ID,
2]            eComponent.entityName Component_Name,
3]           eMainNode.entityName Main_Node_Name
4] FROM       domainMgr d
5] INNER JOIN entity eComponent ON eComponent.domainMgrId = d.domainMgrId
6] INNER JOIN entityData eMainNode ON eMainNode.entityId =
7]                   eComponent.mainNodeEntityId
8] WHERE      eMainNode.entityName = 'VE004.example.net'
9] AND        d.domainName = 'NCOMS';
```

The table below describes this query.

Table 265. Description of the query	
Line numbers	**Description**
1-3	Specify the data to show in the results, as follows: • The entity ID of a component within the specified main node device, represented by `eComponent.entityId`. • The name of a component, represented by `eComponent.entityName`. • The name of the specified main node device, represented by `eMainNode.entityName`.
4	Use the `domainMgr` table as the driving table for this query.
5	Retrieve data for all the entities in this domain by joining the `entity` view to the query. This join retrieves all entities in the domain, including those wholly contained within a single main node (required) as well as those entities related to multiple main nodes , such as VPNs and global VLANs (not required). In this join, the `entity` view is aliased using a meaningful alias, `eComponent`. This alias indicates that the data retrieved from the `entity` view using this join is component data.
6-7	Identify the containing main node device for each of the entities by joining the entityData table to itself using the `mainNodeEntityId` field. This automatically excludes those entities that are related to multiple main node devices, such as VPNs and global VLANs. These entities have a NULL value in the `mainNodeEntityId` field.
8-9	Limit the entities retrieved to those contained within the main node VE004.example.net and the domain to the NCOMS domain.

The table below describes the results of this query.

Table 266. Results of the query		
Component entity ID	**Component name**	**Main node name**
83	`VE004.example.net`	`VE004.example.net`
84	`VLAN_trunk_1_VE004.example.net [0[26]]`	`VE004.example.net`
2151	`VE004.example.net[0[21]]`	`VE004.example.net`
2224	`VE004.example.net[0[14]]`	`VE004.example.net`
2226	`VE004.example.net[0[10]]`	`VE004.example.net`
2227	`VE004.example.net[0[15]]`	`VE004.example.net`
2228	`VE004.example.net[0[12]]`	`VE004.example.net`
2231	`VE004.example.net[0[1]IP:192.168. 32.65]`	`VE004.example.net`
2232	`VE004.example.net[0[13]]`	`VE004.example.net`
2233	`VLAN_OBJECT_VE004.example.net _VLAN_1`	`VE004.example.net`
3187	`VE004.example.net_CARD_0`	`VE004.example.net`

Related reference

Aliasing
Aliasing is the use of a temporary name for a column, sub-query or table within a query.

mainNodeEntityId field
The mainNodeEntityId field in the `entityData` table specifies the main node of the entity. This field provides a shortcut to the main node for a particular entity, avoiding the need to traverse the entire containment tree.

List all components on a device and show component type

This query displays all the components on a device and also displays the *type* of component.

To determine the type of component, the query uses the entityType value for the device. The system object identifier is a numerical key that specifies the type of entity. For example, an entityType value of 1 indicates a main node; an entityType value of 2 indicates an interface.

The entityType table provides a comprehensive list of every entity type in NCIM. This query joins the entityType table to the query to extract the name of the entity type for each component.

Example

```
1] SELECT     eComponent.entityId Component_Entity_ID,
2]            eComponent.entityName Component_Name,
3]            et.typeName Component_Type,
4]            eMainNode.entityName Main_Node_Name
5] FROM       domainMgr d
6] INNER JOIN entity eComponent ON eComponent.domainMgrId = d.domainMgrId
7] INNER JOIN entityData eMainNode ON eMainNode.entityId =
8]                    eComponent.mainNodeEntityId
9] INNER JOIN entityType et ON et.entityType = eComponent.entityType
10] WHERE     eMainNode.entityName = 'VE004.example.net'
11] AND       d.domainName = 'NCOMS';
```

Description

The table below describes how the query determines the type of component.

Table 267. Description of the query

Line numbers	Description
3	In addition to the main node and component data, this query also retrieves the component type of the component contained within the named device. This is represented by `et.typeName`.
9-10	Join the `entityType` table to extract data related to the entity type, including the name of the entity type for each component.

Results

The table below shows the results of this query.

Table 268. Results of the query

Component entity ID	Component name	Component type	Main node name
83	`VE004.example.net`	`chassis`	`VE004.example.net`

Table 268. Results of the query (continued)

Component entity ID	Component name	Component type	Main node name
84	VLAN_trunk_1_VE004.example.net [0[26]]	vlanTrunkEndPoint	VE004.example.net
2151	VE004.example.net[0[21]]	interface	VE004.example.net
2224	VE004.example.net[0[14]]	interface	VE004.example.net
2226	VE004.example.net[0[10]]	interface	VE004.example.net
2227	VE004.example.net[0[15]]	interface	VE004.example.net
2228	VE004.example.net[0[12]]	interface	VE004.example.net
2231	VE004.example.net[0[1] IP:192.168.32.65]	ipEndPoint	VE004.example.net
2232	VE004.example.net[0[13]]	interface	VE004.example.net
2233	VLAN_OBJECT_VE004.example.net _VLAN_1	localVlan	VE004.example.net
3187	VE004.example.net_CARD_0	module	VE004.example.net

Display the number of cards on each device

This query lists all of the main node devices in a domain and retrieves the number of cards in each of these devices.

The query retrieves only devices that contain at least two cards; devices that have no cards are not displayed.

Examples of cards include Three-Port Gigabit Ethernet cards, WAN interface cards, and mainboards cards.

Example

```
1]  SELECT     eMainNode.entityName Main_Node_Entity_Name,
2]            COUNT(physicalCard.entityId) AS 'Number of Cards'
3]  FROM       domainMgr d
4]  INNER JOIN entity eCard ON eCard.domainMgrId = d.domainMgrId
5]  INNER JOIN physicalCard ON physicalCard.entityId = eCard.entityId
6]  INNER JOIN entityData eMainNode ON eMainNode.entityId =
7]                   eCard.mainNodeEntityId
8]  WHERE      d.domainName = 'NCOMS'
9]  GROUP BY   eMainNode.entityId
10] HAVING     count(physicalCard.entityId) > 1
```

Description

The table below describes this query.

Table 269. Description of the query	
Line numbers	**Description**
1-2	Specify the data to show in the results, as follows: • The name of a main node device, represented by `eMainNode.entityname` • The number of cards in that device, represented by `COUNT(physicalCard.entityId)`
4-6	Join relevant tables to the domainMgr table in order to retrieve the required data. The joins are as described in the next two rows.
4	Retrieve all the entities in each domain. The `INNER JOIN` clause ensures that only entities that have a valid `domainMgrId` field are retrieved.
5	From all the entities, extract only that subset of entities that are cards. Use an `INNER JOIN` statement to ensure that only entities that have corresponding entries in the `physicalCard` table are retrieved. There is a line of data for each card. This line of data consists of all columns from the `domain` table, the `entity` view, and the `physicalCard` table related to that card.
6-7	Identify the containing main node device for each of the cards by joining the `entityData` table to itself using the `mainNodeEntityId` field. The join on this field enables the query to go directly to the top of the containment tree.
8	Limit the resulting data to the main node devices in the `NCOMS` domain only.
9	Group the results by the name of the main node device. This means that the results show the number of cards within each main node.
10	Use the HAVING clause to specify that you want to retrieve only devices that contain two or more cards.

Results

The table below shows a portion of the results for this query.

Table 270. Results of the query	
Main node entity name	**Number of cards**
`172.18.1.102`	`20`
`VE001.example.net`	`10`

Related reference

mainNodeEntityId field

The mainNodeEntityId field in the `entityData` table specifies the main node of the entity. This field provides a shortcut to the main node for a particular entity, avoiding the need to traverse the entire containment tree.

Find all devices containing Three-Port Gigabit Ethernet cards

This query looks for specific containment information within a device. In this example, the query finds all main-node devices that contain a specific component: a Cisco Three-Port Gigabit Ethernet card.

This query also returns the following information about each Three-Port Gigabit Ethernet Card retrieved:

- Serial number of the card
- Hardware revision of the card
- The physical position occupied within the main node device by the slot that contains this card

Tip: To perform this type of query, you need to know the MIB OID of the component contained within the device. In this example, you need to know that the OID of the Three-Port Gigabit Ethernet Card within the Cisco MIB is `1.3.6.1.4.1.9.12.3.1.9.18.49`.

Prerequisites

Before you run this query, you must have enabled the Entity agent to run during the discovery process. This enables the query to retrieve the required data. The Entity agent discovers detailed containment information from the Entity MIB. For more information about the Entity agent, see the *IBM Tivoli Network Manager IP Edition Administration Guide*. By default the Entity agent is not configured to run during discovery. You must therefore configure this agent manually if you want the topology database to contain the detailed MIB information that is required for queries of this type.

Example

```
1]  SELECT     d.domainname Domain,
2]             e2.entityName Device_Name,
3]             c.serialnumber Serial_Number,
4]             c.hwRevision Hardware_Revision,
5]             s.relativePosition Slot_Number
6]  FROM       domainmgr d
7]  INNER JOIN entity e1 ON  e1.domainmgrid = d.domainmgrid
8]  INNER JOIN physicalCard c ON c.entityid = e1.entityid
9]  INNER JOIN entityData e2 ON e2.entityid = e1.mainnodeentityid
10] INNER JOIN contains c2 ON c2.containedentityid = c.entityid
11] INNER JOIN physicalSlot s ON s.entityid = c2.containingentityid
12] WHERE      c.vendorType = '1.3.6.1.4.1.9.12.3.1.9.18.49'
13] ORDER BY LOWER(d.domainname) ASC, LOWER(e2.entityName) ASC;
```

Description

The table below describes this query.

Table 271. Description of the query

Line numbers	Description
1-5	Specify the data to show in the results, as follows: • The domain to which the main node device belongs, represented by `d.domainname` • The name of a main node device containing a Three-Port Gigabit Ethernet card, represented by `e2.entityName` • The serial number of the Three-Port Gigabit Ethernet card, represented by `c.serialnumber` • The hardware version of the Three-Port Gigabit Ethernet card, represented by `c.hwRevision` • The slot occupied by the Three-Port Gigabit Ethernet card in the main node device, represented by `s.relativePosition`
7-11	Join relevant tables to the `domainMgr` table in order to retrieve the required data.
7	Retrieve all the entities in each domain. The `INNER JOIN` clause ensures that only entities that have a valid `domainMgrId` field are retrieved.
8	From all the entities, extract only that subset of entities that are cards. Card data is held in the `physicalCard` table. There is a line of data for each card. This line of data consists of all columns from the `domain` table, the `entity` view, and the `physicalCard` tables related to that card.
9	For each card, obtain the name of the main node that contains that card. Do this by performing a second `INNER JOIN` statement on the `entityData` table to retrieve all the data for the main node that contains the card.

Table 271. Description of the query (continued)	
Line numbers	**Description**
10-11	These two lines retrieve for each card, the physical position occupied within the main node device by the slot that contains that card.
10	The query has so far extracted all cards in the database, together with line of relevant data for each card. From all these cards, extract only those cards that are contained within another entity. Do this by performing an `INNER JOIN` statement between the `physicalCard` table and the `contains` table. This `INNER JOIN` statement also retrieves the `containingEntityId` column values, which are the IDs of the entities containing the cards.
11	For each card, obtain data for the slot that contains the card. Do this by performing an `INNER JOIN` statement between the `physicalSlot` table and the `contains` table to retrieve all the data for the main node that contains the card. This limits the results to only those cards which are contained within slots.
12	Limit the resulting data to those cards that have an OID of `1.3.6.1.4.1.9.12.3.1.9.18.49`. This OID corresponds to the MIB variable `cevGsr3ge`, which is the MIB variable for the Cisco Three-Port Gigabit Ethernet Card.
13	For readability purposes, order the results first by domain and then by name of the main node device.

Results

The following table shows an example of the results of this query.

Table 272. Results of the query				
Domain	**Device name**	**Serial number**	**Hardware revision**	**Slot number**
NCOMS	`VE001.example.net`	SAD06A400WY	2.0	3
NCOMS	172.20.4.13	SDK04A70XV4	2.0	4
NCOMS	172.50.0.2	SAD06A300PY	2.0	5

Find entities within all cards

This query retrieves entities contained within all cards. Cards might contain entities of different types, including ports, slots, and sensors. The query lists each of the cards identified and, for each card, lists the entities contained within the card.

This query does not traverse the entire containment tree within the card. Therefore, the query only retrieves components at the top level within the card.

This query uses the `contains` table. This table defines all the containment relationships between entities. Each row in the `contains` table holds a pair of entity identifiers: the containing entity and the contained entity identifier. For each card identified, the query joins to the `contains` table and extracts information about one of the entities contained within that card.

Example

```
1] SELECT     container.entityName Card_Name,
2]            m.cardNumber Card_Number,
3]            part.entityName Contained_Entity
```

```
4] FROM       physicalCard m
5] INNER JOIN entityData container ON container.entityId = m.entityId
6] INNER JOIN contains c ON c.containingEntityId = m.entityId
7] INNER JOIN entityData part ON part.entityId = c.containedEntityId
8] ORDER BY   container.entityName
```

The table below describes this query.

Table 273. Description of the query

Line numbers	Description
1-3	Specify the data to show in the results, as follows: • The name of the card, represented by `container.entityName` • The number of the card within the main node device, represented by `m.cardNumber` • The name of the interface, represented by `part.entityName`
4	Use the `physicalCard` table as the driving table for this query. The `FROM` clause extracts data for all cards.
5	For each card, extract the full set of entity data for that card. This ensures that the entity name of the card is retrieved for display in the query results, as specified in line 1). Use the alias `container` for the `entityData` table to indicate that data extracted using this alias is data for the containing card. Do this by specifying an `INNER JOIN` statement with the `entityData` table.
6	For each card, extract records from the `contains` table on entities contained within that card. Limit the query results to those cards that contain other entities. Do this by specifying an `INNER JOIN` statement with the `contains` table. The query extracts a record from the `contains` table for each entity contained within a given card. Each of these records includes the entity identifier for an entity contained within the card.
7	Extract the full set of entity data for each contained entity. Use the alias `part` for the `entityData` table to indicate that data extracted using this alias is data for a contained entity. Do this by specifying a second `INNER JOIN` statement with the `entityData` table.
8	To facilitate readability of the results, order by the entity name of the containing card.

The table below shows the results of this query.

Table 274. Results of the query

Card name	Card number	Contained entity
`10.1.1.11_CARD_1`	`1`	`10.1.1.11[ 1 [ 1 ] ]`
`10.1.1.11_CARD_2`	`2`	`10.1.1.11[ 2 [ 1 ] ]`
`10.1.1.12_CARD_0`	`0`	`10.1.1.12[ 0 [ 14 ] ]`
`10.1.1.12_CARD_0`	`0`	`10.1.1.12[ 0 [ 10 ] ]`

Table 274. Results of the query (continued)		
Card name	**Card number**	**Contained entity**
10.1.1.12_CARD_0	0	10.1.1.12[0 [12]]
10.1.1.12_CARD_0	0	10.1.1.12[0 [11]]
10.1.1.12_CARD_0	0	10.1.1.12[0 [13]]
10.1.1.8_CARD_I3_R0	NULL	10.1.1.8_SLOT_I4_R0'
10.1.1.8_CARD_I3_R0	NULL	10.1.1.8_SLOT_I6_R1'
10.1.1.9_CARD_I3_R0	NULL	10.1.1.9_SLOT_I4_R0'
10.1.1.9_CARD_I3_R0	NULL	10.1.1.9_SLOT_I6_R1'
10.1.254.2_CARD_I1000_R1	NULL	10.1.254.2_SENSOR_I1002_R2
10.1.254.2_CARD_I1000_R1	NULL	10.1.254.2_SENSOR_I1001_R1
10.1.254.2_CARD_I1100_R1	NULL	10.1.254.2_PORT_I1102_R1
10.1.254.2_CARD_I1100_R1	NULL	10.1.254.2_PORT_I1101_R0

Related reference

Aliasing
Aliasing is the use of a temporary name for a column, sub-query or table within a query.

Table joins
Use table joins to combine records from one or more tables. Two types of table join are used, INNER JOIN and OUTER JOIN.

Choice of driving table
One of the most important design decisions when creating a query is the choice of driving table. The choice of *driving table* is particularly important for ensuring the efficiency of queries.

Queries for port and interface information

These sample queries extract interface and protocol information associated with interfaces.

Device entities, usually interfaces, might be associated with protocol-specific data. The most common example is the association between a device interface with the IP addressing data. Interfaces might also be associated with other types of addressing data, including ATM protocol data and OSPF protocol data.

List all interfaces on all devices

This query provides a list of all main node devices within a domain together with the identifiers and names of the interfaces on each device.

Example

```
1] SELECT     eMainNode.entityName Main_Node_Name,
2]            eInterface.entityId Interface_Entity_ID,
3]            eInterface.entityName Interface_Entity_Name
4] FROM       entityData eInterface
5] INNER JOIN entityData eMainNode ON eMainNode.entityId =
6]                    eInterface.mainNodeEntityId
```

```
 7] WHERE      eInterface.entityType = 2
 8] ORDER BY   eMainNode.entityName, eInterface.entityName
```

Description

The table below describes this query.

Table 275. Description of the query

Line numbers	Description
1-3	Specify the data to show in the results, as follows: • The name of a main node device, represented by `eMainNode.entityName` • The unique entity ID of the interface within the topology database, represented by `eInterface.entityId` • The name of the interface, represented by `eInterface.entityName`
4	Use the `entityData` table as the driving table for this query. Use the alias `eInterface` for the `entityData` table to indicate that the data extracted using this alias is interface data.
5-6	Identify the containing main node device for each of the entities retrieved in the preceding line. Do this by joining the `entityData` table to itself using the `mainNodeEntityId` field.
7	Limit the components of the device to interfaces only. Do this filtering the components to retrieve only components with an entity type of 2, which corresponds to an interface.
8	To facilitate readability of the results, order first by main node name and then by interface name.

Results

The table below shows a portion of the results for this query.

Table 276. Results of the query

Main node name	Interface entity ID	Interface entity name
172.20.1.41	1622	172.20.1.41[0[1]]
172.20.4.11	1621	172.20.4.11[0[1]]
172.20.4.11	1624	172.20.4.11[0[10]]
172.20.4.11	1479	172.20.4.11[0[11]
172.20.4.11	1632	172.20.4.11[0[12]
VE001.example.net	1631	VE001.example.net[0[1]]

Related reference

mainNodeEntityId field

The mainNodeEntityId field in the `entityData` table specifies the main node of the entity. This field provides a shortcut to the main node for a particular entity, avoiding the need to traverse the entire containment tree.

entityType field
The entityType field can be used in SQL queries to limit the type of component data that is retrieved.

List all interfaces with specific attributes

This query provides a list of all interfaces within a domain that have specific attribute values.

The example given here retrieves interfaces that have an interface speed greater than 155 MB per second; however, you can construct a query using any of the attributes in the `networkInterface` table.

Example

```
1] SELECT     e.entityName Interface_Name,
2]            i.ifName IfName,
3]            i.ifSpeed Interface_Speed
4] FROM       entityData e
5] INNER JOIN networkInterface i ON i.entityId = e.entityId
6] WHERE      ifSpeed > 155000000
7] ORDER BY   i.ifSpeed DESC;
```

Description

The table below describes this query.

Table 277. Description of the query

Line numbers	Description
1-3	Specify the data to show in the results, as follows: • The name of an interface, represented by `e.entityName` • The name of the interface stored in the MIB, represented by `i.ifName` • The speed of the interface, represented by `i.ifSpeed`
4	Use the `entityData` table as the driving table for this query. This part of the query retrieves all entities held in the database.
5	Limit the results of the query to interfaces. Do this by joining the `networkInterface` table to the `entityData` table using the `mainNodeEntityId` field. There is now a line of data for each interface in the database. The `INNER JOIN` statement ensures that only interface data is retrieved.
6	Limit the results of the query to interfaces with interface speeds greater than 155 MB per second.
7	Order the results by the speed of the interface.

Results

The table below shows the results of this query.

Table 278. Results of the query		
Interface name	**IfName**	**Interface speed**
10.1.254.2[1 [1]]	Gi1/1	1000000000
192.170.170.10[0 [51]]	Gi50	1000000000
192.170.170.10[0 [50]]	Gi49	1000000000
192.170.170.10[0 [1]]	FX1	1000000000
172.20.4.19[0 [1]]	ATM0/1/0	622080000
172.18.1.102[2 [1]]	FEC-9/39-42	400000000
172.20.4.19[0 [2]]	ATM0	155520000
192.170.170.10[0 [52]]	Co51	155520000

List all interfaces on all devices with interface type

This query retrieves all interfaces on all devices across all domains, and also retrieves information about the interface.

For each interface the query retrieves the following information about the interface:

- `ifName`
- `ifType`
- A textual description corresponding to the `ifType` field

In addition to using information from the `entityData` table to list the interfaces on each device, this query provides a join to the `networkInterface` table to bring in detailed attribute data for the interfaces identified.

Example

```
1]  SELECT     eInterface.entityId Interface_Entity_ID,
2]            eMainNode.entityName Main_Node_Name,
3]            eInterface.entityName Interface_Entity_Name,
4]            i.ifName IfName,
5]            i.ifType Interface_Type,
6]            i.ifTypeString Interface_Type_String
7]  FROM       entityData eInterface
8]  INNER JOIN entityData eMainNode ON eMainNode.entityId =
9]                  eInterface.mainNodeEntityId
10] INNER JOIN networkInterface i ON i.entityId = eInterface.entityId
11] WHERE      eInterface.entityType = 2
12] ORDER BY   eMainNode.entityName, i.ifType
```

Description

The table below describes this query.

Table 279. Description of the query	
Line numbers	**Description**
1-6	Specify the data to show in the results, as follows: • The unique entity ID of the interface within the topology database, represented by `eInterface.entityId` • The name of the main node to which the interface belongs, represented by `eMainNode.entityName` • The name of the interface, represented by `eInterface.entityName` • The textual name of the interface, as specified in the MIB, represented by `i.ifName` • The type of interface, as specified in the MIB, represented by `i.ifType` • The textual description corresponding to this type of interface, represented by `i.ifTypeString`
7	Use the `entityData` table as the driving table for this query. Use the alias `eInterface` for the `entityData` table to indicate that the data extracted using this alias is interface data.
8-9	Identify the containing main node device for each of the entities retrieved in the preceding line. Do this by joining the `entityData` table to itself using the `mainNodeEntityId` field.
10	Extract all attribute data for the various interfaces. This attribute data is held in the `networkInterface` table. Do this by joining the `networkInterface` table to the `entityData` table using the `entityId` field. The `INNER JOIN` statement ensures that only interface data is retrieved.
11	Limit the components of the device to interfaces only. Do this by filtering the components to retrieve only components with an entity type of 2, which corresponds to an interface.
12	To facilitate readability of the results, order first by main node name and then by `ifType`.

Results

The table below shows the results of this query.

Table 280. Results of the query					
Interface entity ID	**Main node name**	**Interface entity name**	**IfName**	**Interface type**	**Interface type string**
1479	10.1.1.11	10.1.1.11[0 [12]	Fa0/11	6	ethernetCsmacd
1621	10.1.1.11	10.1.1.11[0 [10]	Fa0/9	6	ethernetCsmacd
1622	10.1.1.11	10.1.1.11[0 [1]	VL1	6	ethernetCsmacd

Table 280. Results of the query (continued)

Interface entity ID	Main node name	Interface entity name	IfName	Interface type	Interface type string
2466	10.1.1.5	10.1.1.5[0 [1029]	1B1	18	ds1
2471	10.1.1.5	10.1.1.5[0 [1035]]	1B4	18	ds1
2465	10.1.1.5	10.1.1.5[0 [1032]]	1B2	37	atm
2476	10.1.1.5	10.1.1.5[0 [1030]]	1B1	37	atm
2477	10.1.1.5	10.1.1.5[0 [1024]]	1CTL	37	atm
2474	10.1.1.5	10.1.1.5[0 [1059]]		44	frameRelayService
2480	10.1.1.5	10.1.1.5[0 [1053]]		44	frameRelayService
2482	10.1.1.5	10.1.1.5[0 [1055]]		44	frameRelayService
2488	10.1.1.5	10.1.1.5[0 [5]]	qaa1	114	ipOverAtm
2490	10.1.1.5	10.1.1.5[0 [4]]	qaa0	114	ipOverAtm
2496	10.1.1.5	10.1.1.5[0 [6]]	qaa2	114	ipOverAtm
1652	10.1.1.9	10.1.1.9[0 [1]]	Se0/0	22	propPointToPointSeriall
1131	10.1.254.1	10.1.254.1[0 [20]]	Fa0/1.80	135	l2vlan
1130	10.1.254.1	10.1.254.1[0 [22]]	Se1/1:0	166	mpls

Related reference

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

List all interfaces on all devices
This query provides a list of all main node devices within a domain together with the identifiers and names of the interfaces on each device.

entityType field

The entityType field can be used in SQL queries to limit the type of component data that is retrieved.

List all IP addresses and the interfaces that implement them

This query retrieves all interfaces on all main node devices. For each interface, the query lists the IP addresses that the interface implements. An interface can implement multiple IP addresses.

In addition to using information from the `entityData` table to list the interfaces on each device, this query lists the IP addresses implemented by each interface identified. If an interface does not implement an IP address, that interface is not returned by this query.

Example

```
1]  SELECT     eInterface.entityId Interface_Entity_ID,
2]             eMainNode.entityName Main_Node_Name,
3]             eInterface.entityName Interface_Entity_Name,
4]             ip.address IP_Address
5]  FROM       entityData eInterface
6]  INNER JOIN entityData eMainNode ON eMainNode.entityId =
7]                   eInterface.mainNodeEntityId
8]  INNER JOIN protocolEndPoint p ON p.implementingEntityId = eInterface.entityId
9]  INNER JOIN ipEndPoint ip ON ip.entityId = p.endPointEntityId
10] WHERE      eInterface.entityType = 2
11] ORDER BY   eInterface.entityId
```

Description

The table below describes this query.

Table 281. Description of the query

Line numbers	Description
1-4	Specify the data to show in the results, as follows: • The unique entity ID of the interface within the topology database, represented by `eInterface.entityId` • The name of a main node device, represented by `eMainNode.entityName` • The name of the interface, represented by `eInterface.entityName` • An IP address implemented by this interface, represented by `ip.address`
5	Use the `entityData` table as the driving table for this query. Use the alias `eInterface` for the `entityData` table to indicate that the data extracted using this alias is interface data.
6-7	Identify the containing main node device for each of the entities retrieved in the preceding line. Do this by joining the `entityData` table to itself using the `mainNodeEntityId` field.
8-9	Identify the IP addresses implemented by each of the entities identified in line 5 of the query. Do this by performing an `INNER JOIN` statement on the `protocolEndPoint` table to extract the entity ID for any protocol-specific information associated with the entities identified in line 5. Then perform a second `INNER JOIN` statement on the `ipEndPoint` table to limit the protocol-specific information returned by the query to IP information.

Table 281. Description of the query (continued)	
Line numbers	**Description**
10	Limit the components of the device to interfaces only. Do this by filtering the components to retrieve only components with an entity type of 2, which corresponds to an interface.
11	To facilitate readability of the results, order first by the unique entity ID of the interface.

Results

The table below shows the results of this query.

Table 282. Results of the query			
Interface Entity ID	**Main Node Name**	**Interface Entity Name**	**IP Address**
270	172.20.4.11	172.20.4.11[0[5]]	172.50.0.2
338	172.18.1.196	172.18.1.196[0[2]]	172.50.0.3
366	172.18.1.54	172.18.1.54[0[2]]	172.50.0.4
370	172.18.1.54	172.18.1.54[0[1]]	172.50.0.5
373	172.20.4.13	172.20.4.13[0[1]]	172.50.0.6
377	172.20.4.20	172.20.4.20[0[1]]	172.50.0.7
417	192.168.139.7	192.168.139.7[0 [5]]	172.20.11.1
417	192.168.139.7	192.168.139.7[0 [5]]	172.20.1.2

Related reference

List all interfaces on all devices
This query provides a list of all main node devices within a domain together with the identifiers and names of the interfaces on each device.

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

ipEndPoint
The ipEndPoint table represents an IP end point and includes relevant data. The endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

mainNodeEntityId field
The mainNodeEntityId field in the `entityData` table specifies the main node of the entity. This field provides a shortcut to the main node for a particular entity, avoiding the need to traverse the entire containment tree.

Protocol endpoint tables

The protocolEndPoint and ipEndPoint tables can be used in SQL queries to identify the IP addresses that are implemented by the device interfaces.

Queries for connectivity information

These sample queries extract data on connectivity within your network.

Connectivity includes connections between different devices, and VLAN-related connections within the same device. In addition, the NCIM database independently represents connectivity of entities in different layers, so that the connectivity at layer 2 is represented independently of the connectivity at layer 3.

NCIM can model complex connectivity scenarios. For example, within the MPLS VPN realm, NCIM can model the layer 3 connection between a provider-edge (PE) router and multiple customer-edge (CE) routers.

Connectivity information is stored in the `connects` table. This table stores each connection as a single record. However, because two entities are involved in a connection, the order of the connected entities in the `connects` table is random.

For example, the following figures shows the devices that are connected to the main node device VE001.example.net.

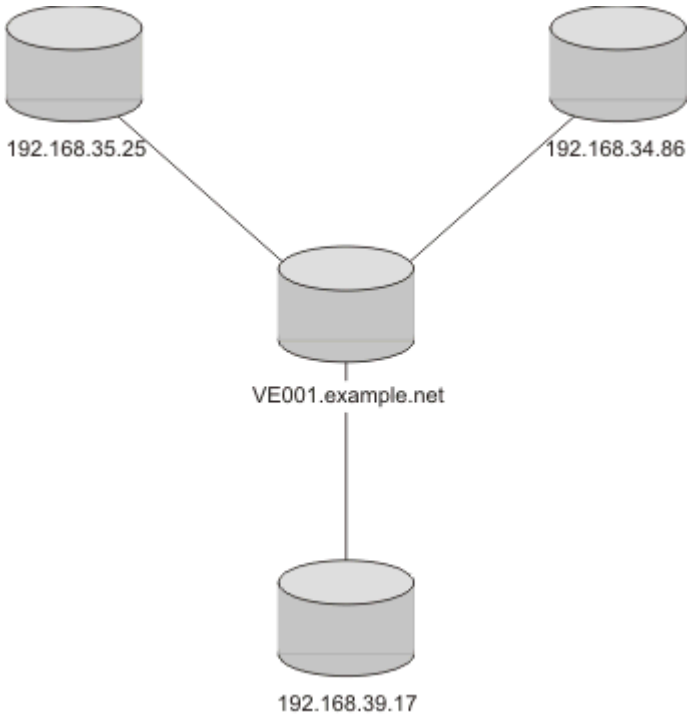


Figure 4. Devices connected to main node device VE001.example.net

The following table shows how the `connects` table might store the data about the connectivity between the device VE001.example.net and neighboring devices.

Table 283. Example data from the connects table for connections to main node device VE001.example.net

connectionId	aEndEntityId	zEndEntityId
101	VE001.example.net	192.168.35.225
102	VE001.example.net	192.168.34.86
103	192.168.39.175	VE001.example.net

It is arbitrary whether a device is designated at the start (the aEnd) or at the end (zEnd) of a connection. The following example from Table 283 on page 521 shows why:

Connections 101 and 102 show the device VE001.example.net at the aEnd of the connection.
Connection 103 shows the device VE001.example.net at the zEnd of the connection.

Connections in NCIM can be bidirectional or unidirectional. A field in the `connects` table that specifies whether the connection is bidirectional or unidirectional.

To ensure that all connections are retrieved from the `connects` table for a given device, the query must take into account the random ordering of aEnd and zEnd data in the table. This is done using a UNION statement. The query works as follows:

```
Find all devices connected to the device VE001.example.net where VE001.example.net
is the aEnd of the connection
UNION
Find all devices connected to the device VE001.example.net where VE001.example.net
is the zEnd of the connection
```

Types of connectivity

Queries that retrieve device connectivity can identify different types of connection. Use this information to learn about the connectivity types that can be queried.

The following types of connectivity are retrieved:

Connections to other devices
The connection passes through a physical or logical interface. Interfaces have an entity type of 2 and are modleled using the `interface` table.

Trunk connection between a specific VLAN on the named device to the same VLAN on a different device
The connection passes through a VLAN trunk port. A VLAN trunk port is a physical port that carries data from multiple VLANs. Each VLAN trunked by the VLAN trunk port is modelled with a VLAN trunk end point.

Connections within the named device between local VLANs and VLAN trunk ports
The connection passes between a local VLAN on the current device to a VLAN trunk on the same device. The query reports this connection as a connection between the device and itself. Local VLANs are modelled using the `localVlan` table.

Related reference

networkInterface
The `networkInterface` table represents interfaces on a chassis device.

vlanTrunkEndPoint
The vlanTrunkEndPoint table represents a VLAN trunk end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

localVlan
The localVlan table specifies which global VLAN the *local VLAN* belongs to. A local VLAN represents all the interfaces on a single chassis device that belong to a global VLAN.

Hierarchy modeling with the networkPipe and pipeComposition tables

The networkPipe table and pipeComposition table can be used together to represent connectivity at different layers, for example the modeling of layer 2 and layer 3 connections.

A layer 3 connection can be considered as a higher-level connection that is defined in terms of lower-level layer 2 connections. A hierarchy of connections is modeled using the networkPipe and pipeComposition tables, as follows:

Rows in the networkPipe table can be combined in collections using the pipeComposition table
The difference between a network pipe and a simple connection is that a network pipe is an entity. This gives the network pipe the following advantages over a simple connection:

- You can associate attributes to the network pipe, for example by using the entityDetails table.
- A network pipe is able to participate in the relationships available to entities, including containment, connectivity, and dependency relationships.

The pipeComposition table allows a higher-level connection to be defined in terms of lower-level connections

The higher-level and lower-level connections are all represented by rows in the networkPipe table.

Find devices connected to a named device

This query identifies all main node devices connected to a single specified main node device.

Example

```
1]  SELECT     locm.entityid Local_Main_Node_Entity_ID,
2]             locm.entityName Local_Main_Node_Entity_Name,
3]             nbrm.entityid Neighbor_Main_Node_Entity_ID,
4]             nbrm.entityName Neighbor_Main_Node_Entity_Name
5]  FROM       entityData loc
6]  INNER JOIN connects c ON c.aEndEntityId = loc.entityId
7]  INNER JOIN entityData nbr ON nbr.entityId = c.zEndEntityId
8]  INNER JOIN entityData nbrm ON nbrm.entityid = nbr.mainnodeentityid
9]  INNER JOIN entityData locm ON locm.entityid = loc.mainnodeentityid
10] WHERE      loc.mainNodeEntityId = 5
11] UNION
12] SELECT     locm.entityid as locMainNodeEntityId,
13]            locm.entityName as locMainNodeEntityName,
14]            nbrm.entityid as nbrMainNodeEntityId,
15]            nbrm.entityName as nbrMainNodeEntityName
16] FROM       entityData loc
17] INNER JOIN connects c ON c.zEndEntityId = loc.entityId
18] INNER JOIN entityData nbr ON nbr.entityId = c.aEndEntityId
19] INNER JOIN entityData nbrm ON nbrm.entityid = nbr.mainnodeentityid
20] INNER JOIN entityData locm ON locm.entityid = loc.mainnodeentityid
21] WHERE      loc.mainNodeEntityId = 5
```

Description

The table below describes this query.

Table 284. Description of the query

Line numbers	Description
1-4	Specify the data to show in the results, as follows: • The unique entity ID of a specified main node device, represented by `locm.entityId`. This is the named device whose neighbors you want to extract from the database. The rest of this description refers to this device as the *local* device • The name of the local device, represented by `locm.entityName` • The unique entity ID of a device that is next to the specified device, represented by `nbrm.entityId` • The name of the neighboring device, represented by `nbrm.entityName`
5	Use the `entityData` table as the driving table for this query. Use the alias `loc` for the `entityData` table to indicate that the data extracted using this alias is for local entities.
6	Identify all the connections for the entities associated with the local device. Do this by joining the `connects` table using the `aEndEntityId` value.

Table 284. Description of the query (continued)	
Line numbers	**Description**
7	Extract the entity data for each neighboring entity. Do this by joining the `entityData` table a second time using the `zEndEntityId` value. Use the alias `nbr` for the `entityData` table to indicate that the data extracted using this alias is for neighboring entities.
8	Limit the results to neighboring main node devices only. Do this by joining the `entityData` table a second time using the `mainNodeEntityId` value. Use the alias `nbrm` for the `entityData` table to indicate that the data extracted using this alias is entity data for a neighboring main node device.
9	Limit the results to local main node devices only. Do this by joining the `entityData` table a second time using the `mainNodeEntityId`. Use the alias `locm` for the `entityData` table to indicate that the data extracted using this alias is entity data for a local main node device.
10	Specify the identity of the local device.
11	Use the UNION statement to ensure that all connections are retrieved.
12-21	This is the same code as line 1-10 with the difference that here the specified device is considered to be the `zend` (see line 17) and the neighboring devices are all considered to be at the `aend` (see line 18).

Results

The table below shows the results of this query. This data includes examples of devices connected to themselves. These are connections within the same device between local VLANs and VLAN trunk ports.

Table 285. Results of the query			
Local main node entity ID	**Local main node entity name**	**Neighbor main node entity ID**	**Neighbor main mode entity name**
5	`VE001.example.net`	83	`192.168.35.225`
5	`VE001.example.net`	2698	`192.168.34.86`
77	`VE002.example.net`	77	`VE002.example.net`
77	`VE002.example.net`	77	`VE002.example.net`
531	`192.168.39.175`	5	`VE001.example.net`

Related concepts

Connectivity data

Connectivity data defines how entities are connected in the network. It includes connections between different devices, and VLAN-related connections within the same device. Connectivity information is stored in the `topologyLinks`, `networkPipe`, and `pipeComposition` tables.

Related reference

connects
The connects table stores data on connectivity between devices. This table belongs to the category *collections*.

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

Related information

Find all devices connected to a named device together with connecting interfaces
This query identifies all main node devices that are connected to a main device, and also retrieves the interface data that is associated with each of those connections.

Find all devices connected to a named device together with connecting interfaces

This query identifies all main node devices that are connected to a main device, and also retrieves the interface data that is associated with each of those connections.

Example

```
1]  SELECT     locm.entityid Local_Main_Node_Entity_ID,
2]             locm.entityName Local_Main_Node_Entity_Name,
3]             loc.entityName Local_Interface_Name,
4]            nbrm.entityid Neighbor_Main_Node_Entity_ID,
5]             nbrm.entityName Neighbor_Main_Node_Entity_Name,
6]             nbr.entityName Neighbor_Interface_Name
7]  FROM       entityData loc
8]  INNER JOIN connects c ON c.aEndEntityId = loc.entityId
9]  INNER JOIN entityData nbr ON nbr.entityId = c.zEndEntityId
10] INNER JOIN entityData nbrm ON nbrm.entityid = nbr.mainnodeentityid
11] INNER JOIN entityData locm ON locm.entityid = loc.mainnodeentityid
12] WHERE      loc.mainNodeEntityId = 5
13] UNION
14] SELECT     locm.entityid as locMainNodeEntityId,
15]            locm.entityName as locMainNodeEntityName,
16]            loc.entityName as locEntityName,
17]            nbrm.entityid as nbrMainNodeEntityId,
18]            nbrm.entityName as nbrMainNodeEntityName,
19]            nbr.entityName as nbrEntityName
20] FROM       entityData loc
21] INNER JOIN connects c ON c.zEndEntityId = loc.entityId
22] INNER JOIN entityData nbr ON nbr.entityId = c.aEndEntityId
23] INNER JOIN entityData nbrm ON nbrm.entityid = nbr.mainnodeentityid
24] INNER JOIN entityData locm ON locm.entityid = loc.mainnodeentityid
25] WHERE      loc.mainNodeEntityId = 5
```

Description

The table below describes this query.

Table 286. Description of the query	
Line number(s)	**Description**
1-6	Specify the data to show in the results, as follows: • The unique entity ID of a specified main node device. This is the named device whose neighbors you want to extract from the database. The rest of this description refers to this device as the *local* device, represented by `locm.entityId` • The name of the local device, represented by `locm.entityName` • The name of the interface on the local device, represented by `loc.entityName` • The unique entity ID of a device that is adjacent to the specified device, represented by `nbrm.entityId` • The name of the neighboring device, represented by `nbrm.entityName` • The name of the interface on the neighboring device, represented by `nbr.entityName`
7	Extract the entity data for each neighboring entity. Do this by joining the `entityData` table a second time using the `zEndEntityId` value. Use the alias `nbr` for the `entityData` table to indicate that the data extracted using this alias is for neighboring entities.
8	Limit the results to neighboring main node devices only. Do this by joining the `entityData` table a second time using the `mainNodeEntityId` value. Use the alias `nbrm` for the `entityData` table to indicate that the data extracted using this alias is entity data for a neighboring main node device.
9	Limit the results to local main node devices only. Do this by joining the `entityData` table a second time using the `mainNodeEntityId`. Use the alias `locm` for the `entityData` table to indicate that the data extracted using this alias is entity data for a local main node device.
10	Specify the identity of the local device.
11	Use the UNION statement to to ensure that all connections are retrieved.
12-21	This is the same code as line 1-10 with the difference that here the specified device is considered to be the `zend` (see line 21) and the neighboring devices are all considered to be at the `aend` (see line 22).

Results

The following table shows an example of the results of the query.the results of the query.

Table 287. Results of the query

Local main node entity ID	Local main node entity name	Local interface name	Neighbor main node entity ID	Neighbor main node entity name	Neighbor interface name
5	VE001.example.net	VE001.example.net[0[3]]	83	192.168.35.225	192.168.35.225
5	VE001.example.net	VE001.example.net[0[4]]	2698	192.168.34.86	192.168.34.86
77	VE002.example.net	VLAN_OBJECT_VE002.example.net_VLAN_400	77	VE002.example.net	VLAN_trunk_400_VE002.example.net[0 [2]]
77	VE002.example net	VLAN_OBJECT_VE002.example.net_VLAN_1	77	VE002.example.net	VLAN_trunk_1_VE002.example.net[0 [2]]
531	192.168.39.175	192.168.39.175	5	VE001.example.net	VE001.example.net[0[5]]

Identify all connections between routers

This query identifies all connections between routers. These types of connections are also called *Layer 3* router links. Each of these connections also represents a connection between two subnets.

You can use similar queries to determine the type of connection between two devices. You can determine whether a connection falls into any of the following types:

- Layer 2 connection
- Layer 3 router links

  This refers to connections between routers, and hence, between subnets, and is the example provided in this query.
- Psuedowire connection

Use the topologyLinks table to identify which connections belong to a specific type of topology. This table lists all the connections in the database and specifies the identifier of a topology type entity from the `entityData` table.

Example

```
1]  SELECT       a.entityName Connected_Entity,
2]               z.entityName Connected_Entity
3]  FROM         topologyLinks t
4]  INNER JOIN   entityData topo ON topo.entityId = t.entityId
5]  INNER JOIN   connects c ON c.connectionId = t.connectionId
6]  INNER JOIN   entityData a ON a.entityId = c.aEndEntityId
7]  INNER JOIN   entityData z ON z.entityId = c.zEndEntityId
8]  WHERE        topo.entityType = 73
```

Description

The table below describes this query.

Table 288. Description of the query	
Line numbers	**Description**
1-2	Specify the data to show in the results, as follows: • The name of an interface at one end of the connection, represented by `a.entityName` • The name of an interface at the other end of the connection, represented by `z.entityName`
3	Use the `topologyLinks` table as the driving table for this query. Use the alias `t` for the `topologyLinks` table for purposes of brevity.
4	Identify all the types of topology listed in the `topologyLinks` table. Do this by joining the `entityData` table using the `entityId` field.
5	Extract the connection data for each connection. Do this by joining the `connects` table using the `connectionId` field.
6-7	Extract entity details for each of the interfaces at either end of the connection. Do this by joining the `entityData` table a second time using the `entityId` field in the `entityData` table and the `aEndEntityId` and `zEndEntityId` fields in turn in the `connects` table.
8	Limit the results to connections within layer 3 router links only. This limits the results to connections between routers, and hence, between subnets.

Results

The table below shows the results of the query.

Table 289. Results of the query	
Connected entity	**Connected entity**
`172.20.4.16[ Et0/0 ]`	`172.20.4.11[ Fa0/0 ]`
`172.20.4.11[ Fa0/0 ]`	`172.20.4.16[ Et0/0 ]`
`172.20.4.16[ Et0/0 ]`	`172.20.4.12[ Fa0/0 ]`
`172.20.4.11[ Fa0/0 ]`	`172.20.4.12[ Fa0/0 ]`
`172.20.4.12[ Fa0/0 ]`	`172.20.4.16[ Et0/0 ]`
`172.20.4.12[ Fa0/0 ]`	`172.20.4.11[ Fa0/0 ]`
`172.20.4.16[ Et0/0 ]`	`172.20.4.15[ Fa0/1 ]`
`172.20.4.11[ Fa0/0 ]`	`172.20.4.15[ Fa0/1 ]`
`172.20.4.12[ Fa0/0 ]`	`172.20.4.15[ Fa0/1 ]`
`172.20.4.15[ Fa0/1 ]`	`172.20.4.12[ Fa0/0 ]`

Table 289. Results of the query (continued)	
Connected entity	**Connected entity**
172.20.4.15[Fa0/1]	172.20.4.11[Fa0/0]
172.20.4.15[Fa0/1]	172.20.4.16[Et0/0]
172.20.4.16[Et0/0]	172.20.4.28[Gi0/0]

Related reference

Techniques used in the SQL queries
The SQL query examples use a variety of techniques that are aimed at extracting information efficiently. Use this information to familiarize yourself with the techniques used in SQL queries.

Queries for LTE network information

These sample queries retrieve information about Long-Term Evolution (LTE) devices.

Find specific LTE entity types

These queries retrieve details of specific entity types used in LTE networks; for example, Extended NodeB entities, Packet Gateway entities, or Serving Gateway entities.

Example: find all discovered Extended NodeB entities

This query retrieves the details of all Extended NodeB entities that have been discovered.

```
select e.entityId,
 e.entityName,
 enb.eNodeBId,
 enb.eNodeBName,
 enb.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.enbFunction enb ON enb.entityId = e.entityId
```

The table below describes this query.

Table 290. Description of the query	
Line numbers	**Description**
1-5	Specify the data to show in the results, as follows: • The entity ID of the Extended NodeB function entities, represented by `e.entityId`. • The name of the eNodeB, represented by `e.entityName`. • The identifier of the eNodeB, represented by `enb.eNodeBId`. • A more user-friendly name for the eNodeB, represented by `enb.eNodeBName`. • The name by which the enbFunction is known to its element management system (EMS), represented by `enb.emsDistinguishedName`.
6	Use the `entityData` table as the driving table for this query. This part of the query retrieves all entities held in the database.
7	Limit the results of the query to eNodeB function entities. Do this by joining the `enbFunction` table to the `entityData` table using the `entityId` field.

Similar queries

The following example queries retrieve relevant data for different LTE entity types, using similar syntax to the above example.

Example: find all discovered Equipment Identity Register entities

This query retrieves the details of all Equipment Identity Register (EIR) entities that have been discovered.

```
select e.entityId,
 e.entityName,
 eir.eirFunctionName,
 eir.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.eirFunction eir ON eir.entityId = e.entityId
```

Example: find all discovered Home Subscriber Server entities

This query retrieves the details of all Home Subscriber Server (HSS) entities that have been discovered.

```
select e.entityId,
 e.entityName,
 hss.hssFunctionName,
 hss.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.hssFunction hss ON hss.entityId = e.entityId
```

Example: find all discovered Mobility Management Entities

This query retrieves the details of all Mobility Management Entities (MMEs) that have been discovered.

```
select e.entityId,
 e.entityName,
 mme.MMEGI,
 mme.MMEC,
 mme.mmeName,
 mme.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.mmeFunction mme ON mme.entityId = e.entityId
```

Example: find all discovered Packet Gateway entities

This query retrieves the details of all Packet Gateway entities that have been discovered.

```
select e.entityId,
 e.entityName,
 pgw.pgwFunctionName,
 pgw.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.pgwFunction pgw ON pgw.entityId = e.entityId
```

Example: find all discovered Policy and Charging Rule Function entities

This query retrieves the details of all Policy and Charging Rule Function entities that have been discovered.

```
select e.entityId,
 e.entityName,
 pcrf.pcrfFunctionName,
 pcrf.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.pcrfFunction pcrf ON pcrf.entityId = e.entityId
```

Example: find all discovered Serving Gateway entities

This query retrieves the details of all Serving Gateway entities that have been discovered.

```
select e.entityId,
 e.entityName,
 sgw.sgwFunctionName,
 sgw.emsDistinguishedName
from ncim.entityData e
INNER JOIN ncim.sgwFunction sgw ON sgw.entityId = e.entityId
```

Queries for MPLS Traffic Engineered Tunnel information

These sample queries retrieve information about the MPLS Traffic Engineered tunnels that have been discovered.

List all Traffic Engineered tunnels

This database query shows the names of the Traffic Engineered (TE) tunnels that have been discovered, and the domain they are associated with.

Example

```
1] SELECT      e.entityName, e.displayLabel, d.domainName
2] FROM        entityData e
3] INNER JOIN  entityType t on t.entityType = e.entityType
4] INNER JOIN  domainMgr d on d.domainMgrId = e.domainMgrId
5] WHERE       t.typeName = 'MPLS TE Tunnel';
```

Results

The following table provides an example of part of the result set for this query.

Table 291. Results of the query

entityName	displayLabel	domainName
172.20.1.6_MPLS_TE_Tunnel_Idx_10_Inst_0	172.20.1.6 Tunnel10 10:0	NCOMS
172.20.1.6_MPLS_TE_Tunnel_Idx_10_Inst_12	172.20.1.6 Tunnel10 10:12 Primary	NCOMS
172.20.1.7_MPLS_TE_Tunnel_Idx_12_Inst_0	172.20.1.7 Tunnel12 12:0	NCOMS
172.20.1.7_MPLS_TE_Tunnel_Idx_12_Inst_13	172.20.1.7 Tunnel12 12:13 Primary	NCOMS
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_0	172.20.1.7 Tunnel50 50:0	NCOMS
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.7 Tunnel50 50:12 Primary	NCOMS

Show interfaces utilized by Traffic Engineered tunnels

This database query shows the interfaces and physical ports used by a particular Traffic Engineered (TE) tunnel.

Example

```
1] SELECT eTun.entityName as Tunnel, eInt.entityName as Interface
3] FROM collects c
4] INNER JOIN entityData eTun ON eTun.entityId = c.collectingEntityId
5] INNER JOIN entityData eInt ON eInt.entityId = c.collectedEntityId
6] WHERE eTun.entityName = '172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12';
```

Results

The following table provides an example of part of the result set for this query.

Table 292. Results of the query

Tunnel	Interface
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.7[0 [22]]
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.7[0 [24]]
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.4[0 [18]]
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.4[0 [2]]
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.6[0 [2]]
172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12	172.20.1.6[0 [26]]

Show Traffic Engineered tunnel configuration

This query shows a subset of the tunnel attributes for a particular tunnel.

Example

```
1] SELECT e.entityName, m.role, m.ingressLSRId, m.egressLSRId, m.signallingProtocol
2] FROM mplsTETunnel m
3] INNER JOIN entityData e on e.entityId = m.entityId
4] WHERE e.entityName = '172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12';
```

Results

The following table provides an example of the result set for this query.

Table 293. Results of the query

entityName	172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12

Table 293. Results of the query (continued)	
role	head
ingressLSRId	172.20.1.7
egressLSRId	172.20.1.6
signallingProtocol	rsvp

List supporting routers for a Traffic Engineered tunnel

These queries show which routers and services support a particular tunnel.

Example: which router and service support a particular tunnel

```
1] SELECT eHost.entityName as HostingRouter, eServ.entityName as TunnelService,
eTun.entityName as TunnelName
2] FROM entityData eTun
3] INNER JOIN contains c ON c.containedEntityId = eTun.entityId
4] INNER JOIN entityData eServ ON eServ.entityId = c.containingEntityId
5] INNER JOIN hostedService h ON h.hostedEntityId = eServ.entityId
6] INNER JOIN entityData eHost ON eHost.entityId = h.hostingEntityId
7] WHERE eTun.entityName = '172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12';
```

Results

The following table provides an example of part of the result set for this query.

Table 294. Results of the query		
HostingRouter	**TunnelService**	**TunnelName**
172.20.1.7	MPLS_TE_Service_172.20.1.7	172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12

Example: show all routers in a tunnel path

```
1] SELECT DISTINCT eMain.entityName
2] FROM collects c
3] INNER JOIN entityData eTun ON eTun.entityId = c.collectingEntityId
4] INNER JOIN entityData eInt ON eInt.entityId = c.collectedEntityId
5] INNER JOIN entityData eMain ON eMain.entityId = eInt.mainNodeEntityId
6] WHERE eTun.entityName = '172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12';
```

Results

The following table provides an example of part of the result set for this query.

Table 295. Results of the query
entityName
172.20.1.7
172.20.1.4
172.20.1.6

Show performance data for a Traffic Engineered tunnel

This query shows performance data for a tunnel.

Example

```
1] SELECT eTun.entityName, res.maxRate, res.meanRate, res.maxBurstSize,
res.meanBurstSize
2] FROM entityData eTun
3] INNER JOIN dependency d ON d.dependentEntityId = eTun.entityId
4] INNER JOIN mplsTETunnelResource res ON res.entityId = d.independentEntityId
5] WHERE eTun.entityName = '172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12';
```

Results

The following table provides an example of part of the result set for this query.

Table 296. Results of the query

entityName	172.20.1.7_MPLS_TE_Tunnel_Idx_50_Inst_12
maxRate	100000
meanRate	100000
maxBurstSize	1000
meanBurstSize	1

Queries for Radio Access Network information

These sample queries retrieve information about Radio Access Network (RAN) devices.

Find specific RAN entity types

These queries retrieve details of specific entity types used in Radio Access Networks, for example, base stations, node B entities, or Mobile Switching Centres.

Example: find all discovered base stations

This query retrieves the details of all base stations that have been discovered.

```
select e.entityId,
 e.entityName,
 c.className,
 rbs.baseStationId,
 rbs.ranTechnologyType
from ncim.entityData e
INNER JOIN ncim.physicalChassis c ON c.entityId = e.entityId
INNER JOIN ncim.ranBaseStation rbs ON rbs.entityId = c.entityId
```

The table below describes this query.

Table 297. Description of the query	
Line numbers	**Description**
1-5	Specify the data to show in the results, as follows: • The entity ID of the base station, represented by `e.entityId`. • The name of the base station, represented by `e.entityName`. • The Active Object Class assigned to the base station, represented by `c.className`. • The unique identifier of the base station, represented by `rbs.baseStationId`. • The type of wireless technology used by the base station, represented by `rbs.ranTechnologyType`.
6	Use the `entityData` table as the driving table for this query. This part of the query retrieves all entities held in the database.
7	Limit the results of the query to chassis entities. Do this by joining the `chassis` table to the `entityData` table using the `entityId` field.
8	Limit the results of the query to entities that are present in the `ranBaseStation` table, that is, to base stations.

Similar queries

The following example queries retrieve relevant data for different RAN entity types, using similar syntax to the above example.

Example: find all discovered Node B entities

This query retrieves the details of all Node B entities that have been discovered.

```
select e.entityId,
 e.entityName,
 c.className,
 rnb.nodebId,
 rnb.ranTechnologyType
from ncim.entityData e
INNER JOIN ncim.physicalChassis c ON c.entityId = e.entityId
INNER JOIN ncim.ranNodeB rnb ON rnb.entityId = c.entityId
```

Example: find all discovered Base Station Controllers

This query retrieves the details of all base station controllers that have been discovered.

```
select e.entityId,
 e.entityName,
 c.className,
 rbsc.baseStationControllerId,
 rbsc.ranTechnologyType
from ncim.entityData e
INNER JOIN ncim.physicalChassis c ON c.entityId = e.entityId
INNER JOIN ncim.ranBaseStationController rbsc ON rbsc.entityId = c.entityId
```

Example: find all discovered Radio Network Controllers

This query retrieves the details of all Radio Network Controllers that have been discovered.

```
select e.entityId,
 e.entityName,
 c.className,
```

```
 rrnc.rncId,
 rrnc.ranTechnologyType
from ncim.entityData e
INNER JOIN ncim.physicalChassis c ON c.entityId = e.entityId
INNER JOIN ncim.ranRadioNetworkController rrnc ON rrnc.entityId = c.entityId
```

Example: find all discovered Mobile Switching Centers

This query retrieves the details of all Mobile Switching Centers that have been discovered.

```
select e.entityId,
 e.entityName,
 c.className,
 rmsc.mscId,
 rmsc.msctype,
 rmsc.ranTechnologyType
from ncim.entityData e
INNER JOIN ncim.physicalChassis c ON c.entityId = e.entityId
INNER JOIN ncim.ranMobileSwitchingCentre rmsc ON rmsc.entityId = c.entityId
```

Retrieve RAN connectivity

These queries retrieve the details of entities that are connected to RAN entities.

Example: connectivity of a given base station

This query retrieves the connectivity of a given base station.

```
select e1.entityName BTSName,
 e2.entityName BTSConnectedName,
 e3.entityName ConnectedInt,
 e4.entityName ConnectedDevice,
 e4.entityType, ch4.className,
 et.entityName Topology
```

The table below describes this query.

Table 298. Description of the query

Line numbers	Description
1-6	Specify the data to show in the results, as follows: • Name of the base station, represented by `as e1.entityName` • Name of the connected base station, represented by `e2.entityName` • Name of the connected interface, represented by `e3.entityName` • Name of the connected device, represented by `e4.entityName` • Class of the connected device, represented by `e4.entityType` • Name of the connection, represented by `et.entityName`
7-15	Retrieve the data from the following tables: • `entityData` • `ranBaseStation` • `connects` • `topologyLinks` • `chassis`
18	The entity name of the base station is `BaseStation10`.
20	Ensure that the entity is a base station.

Table 298. Description of the query (continued)	
Line numbers	**Description**
22	Limit the results (connected devices) to entities that are main nodes.
24	Identify all the connections for the entities associated with the specified base station.
26	Extract the entity data for each neighboring entity.
28	Determine the connecting point on the other device for the connection. This is captured in `e3.entityId`.
30	Determine the layer in which the other connection is located. This is determined using the `topologyLinks` object.
32	Determine the `entityData` entry corresponding to the topology layer. This enables the query results to specify in which layer the connecting point on the other device is; for example, layer 1,or layer 2.
34	Determine the chassis that the connecting point is in.
36	Use the `UNION` statement to ensure that all connections are retrieved.
37-71	This is the same code as line 1-36 with the difference that here the specified device is considered to be the zend (see line 60) and the neighboring devices are all considered to be at the aend (see line 62).

Similar queries

The following example queries retrieve relevant data for different RAN relationships, using similar syntax to the above example.

Example: connectivity of a given Node B entity

This query retrieves the connectivity of a given Node B entity.

```
select e1.entityName AS NodeBName,
 e2.entityName AS NodeBConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType, ch4.className
from ncim.entityData e1,
 ncim.ranNodeB rnb,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4, ncim.physicalChassis ch4
where
(
    e1.entityName = 'NodeB10'
    AND
    e1.entityId = rnb.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.aEndEntityId
    AND
    e3.entityId = c.zEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
```

```
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
UNION
select e1.entityName AS NodeBName,
 e2.entityName AS NodeBConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranNodeB rnb,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'NodeB10'
    AND
    e1.entityId = rnb.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.zEndEntityId
    AND
    e3.entityId = c.aEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
```

Example: connectivity of a given base station controller

This query retrieves the connectivity of a given base station controller.

```
select e1.entityName AS BSCName,
 e2.entityName AS BSCConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranBaseStationController rbsc,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'BaseStationController2'
    AND
    e1.entityId = rbsc.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.aEndEntityId
    AND
    e3.entityId = c.zEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
```

```
    ch4.entityId = e4.entityId
)
UNION
select e1.entityName AS BSCName,
 e2.entityName AS BSCConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType, ch4.className
from ncim.entityData e1,
 ncim.ranBaseStationController rbsc,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'BaseStationController2'
    AND
    e1.entityId = rbsc.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.zEndEntityId
    AND
    e3.entityId = c.aEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
```

Example: connectivity of a given radio network controller

This query retrieves the connectivity of a given radio network controller.

```
select e1.entityName AS RNCName,
 e2.entityName AS RNCConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranRadioNetworkController rrnc,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'radioNetworkController2'
    AND
    e1.entityId = rrnc.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.aEndEntityId
    AND
    e3.entityId = c.zEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
UNION
select e1.entityName AS RNCName,
```

```
 e2.entityName AS RNCConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranRadioNetworkController rrnc,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'radioNetworkController2'
    AND
    e1.entityId = rrnc.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.zEndEntityId
    AND
    e3.entityId = c.aEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
```

Example: connectivity of a given media gateway

This query retrieves the connectivity of a given media gateway.

```
select e1.entityName AS MGWName,
 e2.entityName AS MGWConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranMediaGateway rmgw,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'MediaGateway1'
    AND
    e1.entityId = rmgw.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.aEndEntityId
    AND
    e3.entityId = c.zEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
UNION
select e1.entityName AS MGWName,
 e2.entityName AS MGWConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
```

```
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranMediaGateway rmgw,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'MediaGateway1'
    AND
    e1.entityId = rmgw.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.zEndEntityId
    AND
    e3.entityId = c.aEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
```

Example: connectivity of a given serving GPRS support node

This query retrieves the connectivity of a given serving GPRS support node.

```
select e1.entityName AS SGSNName,
 e2.entityName AS SGSNConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType, ch4.className
from ncim.entityData e1,
 ncim.ranSGSN rsgsn,
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'SGSN3'
    AND
    e1.entityId = rsgsn.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.aEndEntityId
    AND
    e3.entityId = c.zEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
UNION
select e1.entityName AS SGSNName,
 e2.entityName AS SGSNConnectedName,
 e3.entityName AS ConnectedInt,
 e4.entityName AS ConnectedDevice,
 e4.entityType,
 ch4.className
from ncim.entityData e1,
 ncim.ranSGSN rsgsn,
```

```
 ncim.entityData e2,
 ncim.entityData et,
 ncim.topologyLinks tl,
 ncim.connects c,
 ncim.entityData e3,
 ncim.entityData e4,
 ncim.physicalChassis ch4
where
(
    e1.entityName = 'SGSN3'
    AND
    e1.entityId = rsgsn.entityId
    AND
    e2.mainNodeEntityId = e1.entityId
    AND
    e2.entityId = c.zEndEntityId
    AND
    e3.entityId = c.aEndEntityId
    AND
    c.connectionId = tl.connectionId
    AND
    tl.entityId = et.entityId
    AND
    e3.mainNodeEntityId = e4.entityId
    AND
    ch4.entityId = e4.entityId
)
```

Find RAN containment

These queries retrieve the details of RAN entities that are contained, by other entities.

Example: find all sectors within a given cell

This query retrieves the details of all sectors within a given cell. There is no direct relationship between a sector and a cell. Sectors are hosted by transceivers, and transceivers are contained within a base station. There is a collects relationship between cells and transceivers. The query also deals with fact that there are two different types of cell: GSM cells and UTRAN cells.

```
select e1.entityId sectorEntityId,
 e1.entityName sectorName,
 e2.entityId cellEntityId ,
 e2.entityName cellEntityName,
 COALESCE(rgc.cellid, ruc.cellid),
 COALESCE(rgc.rantechnologytype,'UMTS') cellType
from ncim.entityData e1
INNER JOIN ncim.ranSector rs ON rs.entityId = e1.entityId
INNER JOIN ncim.hostedService hs ON hs.hostedEntityId = e1.entityId
INNER JOIN ncim.entityData e3 ON e3.entityId = hs.hostingEntityId
INNER JOIN ncim.collects c ON c.collectedEntityId = e3.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = c.collectingEntityId
LEFT OUTER JOIN ncim.rangsmcell rgc ON rgc.entityId = e2.entityId
LEFT OUTER JOIN ncim.ranutrancell ruc ON ruc.entityId = e2.entityId
WHERE
(
    e2.entityName = cell_name
    AND
    (
        e2.entityType = 130
        OR
        e2.entityType = 131
    )
);
```

The table below describes this query.

Table 299. Description of the query	
Line numbers	**Description**
1-6	Specify the data to show in the results, as follows: • The entity ID of the sector, represented by `e1.entityId`. • The name of the sector, represented by `e1.entityName` • The ID of the cell, represented by `e2.entityId` • The name of the cell, represented by `e2.entityName` • Use the `COALESCE` function to take either GSM or UTRAN cell IDs as a return value.
7	Use the `entityData` table as the driving table for this query.
8	Limit the results of the query to RAN sectors
9-10	The alias e3 identifies the hosting transceiver. The JOIN operations on these lines limit the results to the transceiver that hosts the RAN sectors.
11-12	The alias e2 identifies the cells that collect the transceivers. The JOIN operations on these lines limit the results to the cells that collect the transceiver, that in turn hosts the RAN sectors.
13-14	Join the two cell tables, GSM and UTRAN. Use an outer join, as one of these tables will be empty.
15-23	Specify the cell name and include results for GSM cells (`entityType = 130`) and UTRAN cells (`entityType = 131`).

Similar queries

The following example queries retrieve relevant data for different RAN relationships, using similar syntax to the above example.

Example: find contents of the RAN radio core

This query retrieves the contents of the RAN radio core.

```
SELECT e.entityId,
 e.entityName, ch.className,
 e2.entityName RANRadioCore,
 rrc.mmc, rrc.mnc
FROM ncim.entityData e
INNER JOIN ncim.physicalChassis ch ON ch.entityId = e.entityId
INNER JOIN ncim.collects c ON c.collectedEntityId = e.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = c.collectingEntityId
INNER JOIN ncim.ranRadioCore rrc ON rrc.entityId = e2.entityId
WHERE
e2.entityType = 138
```

Example: find contents of the RAN circuit-switched core

This query retrieves the contents of the RAN circuit-switched core.

```
SELECT e.entityId,
 e.entityName, ch.className,
 e2.entityName RANCircuitSwitchedCore,
 rcsc.mmc, rcsc.mnc
FROM ncim.entityData e
INNER JOIN ncim.physicalChassis ch ON ch.entityId = e.entityId
INNER JOIN ncim.collects c ON c.collectedEntityId = e.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = c.collectingEntityId
```

```
INNER JOIN ncim.ranCircuitSwitchedCore rcsc ON rcsc.entityId = e2.entityId
WHERE
e2.entityType = 137
```

Example: find contents of the RAN packet-switched core

This query retrieves the contents of the RAN packet-switched core.

```
SELECT e.entityId,
 e.entityName, ch.className,
 e2.entityName RANPacketSwitchedCore,
 rpsc.mmc,
 rpsc.mnc
FROM ncim.entityData e
INNER JOIN ncim.physicalChassis ch ON ch.entityId = e.entityId
INNER JOIN ncim.collects c ON c.collectedEntityId = e.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = c.collectingEntityId
INNER JOIN ncim.ranPacketSwitchedCore rpsc ON rpsc.entityId = e2.entityId
WHERE
e2.entityType = 136
```

Find RAN dependencies

These queries retrieve the details of RAN entities that are dependent upon other RAN entities.

Example: find all cells managed by a given base station

This query retrieves the details of all cells that are managed by a given base station.

```
select e1.entityId cellEntityId,
 e1.entityName cellName,
 rc.cellid,
 e2.entityId btsEntityId,
 e2.entityName BTSname,
 rbs.baseStationId
from ncim.entityData e1
INNER JOIN ncim.rangsmcell rc ON rc.entityId = e1.entityId
INNER JOIN ncim.dependency dep ON dep.dependentEntityId = e1.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = dep.independentEntityId
INNER JOIN ncim.ranBaseStation rbs ON rbs.entityId = e2.entityId
WHERE (e2.entityName = base_station_name)
```

The table below describes this query.

Table 300. Description of the query

Line numbers	Description
1-6	Specify the data to show in the results, as follows: • The entity ID of the cell, represented by `e1.entityId`. • The name of the cell, represented by `e1.entityName` • The ID of the cell, represented by `rc.cellid` • The entity ID of the base station, represented by `e2.entityId` • The name of the base station, represented by `e2.entityName` • The identifying string of the base station, represented by `rbs.baseStationId`
7	Use the `entityData` table as the driving table for this query.
8	Limit the results of the query to RAN GSM cells. Do this by joining the `ranGSMCell` table to the `entityData` table using the `entityId` field.

Table 300. Description of the query (continued)	
Line numbers	**Description**
9-11	Limit the results of the query to cells that have dependencies listed on entities that are RAN base stations.
12	Limit the results of the query to cells managed by the base station known as *base_station_name*.

Similar queries

The following example queries retrieve relevant data for different RAN relationships, using similar syntax to the above example.

Example: find all cells managed by a given Node B entity

This query retrieves the details of all cells managed by a given Node B entity.

```
select e1.entityId cellEntityId,
 e1.entityName cellName, rc.cellid,
 e2.entityId nodeBEntityId,
 e2.entityName nodeBName,
 rnb.nodeBId
from ncim.entityData e1
INNER JOIN ncim.ranutrancell rc ON rc.entityId = e1.entityId
INNER JOIN ncim.dependency dep ON dep.dependentEntityId = e1.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = dep.independentEntityId
INNER JOIN ncim.ranNodeB rnb ON rnb.entityId = e2.entityId
WHERE
(
    e2.entityName = node_b_name
)
```

Example: find all base stations managed by a given base station controller

This query retrieves the details of all base stations managed by a given base station controller.

```
select e1.entityId btsEntityId,
 e1.entityName btsName,
 rbs.basestationid,
 e2.entityId bscEntityId,
 e2.entityName bscName,
 rbsc.baseStationControllerId
from ncim.entityData e1
INNER JOIN ncim.ranBaseStation rbs ON rbs.entityId = e1.entityId
INNER JOIN ncim.dependency d ON d.dependentEntityId = e1.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = d.independentEntityId
INNER JOIN ncim.ranBaseStationController rbsc ON rbsc.entityId = e2.entityId
WHERE
(
    e2.entityName = base_station_controller_name
);
```

Example: find all Node B entities managed by a given radio network controller

This query retrieves the details of all Node B entities managed by a given radio network controller.

```
select e1.entityId nodeBEntityId,
 e1.entityName nodeBName,
 rnb.nodeBid,
 e2.entityId rncEntityId,
 e2.entityName rncName,
 rrnc.rncId
from ncim.entityData e1
INNER JOIN ncim.ranNodeB rnb ON rnb.entityId = e1.entityId
INNER JOIN ncim.dependency d ON d.dependentEntityId = e1.entityId
INNER JOIN ncim.entityData e2 ON e2.entityId = d.independentEntityId
```

```
INNER JOIN ncim.ranRadioNetworkController rrnc ON rrnc.entityId = e2.entityId
WHERE
(
    e2.entityName = radio_network_controller_name
);
```

Queries for hosted services

These queries extract data on services or applications running on specific devices.

A hosted service is a service or application running on a specific device. For example, a device can host BGP or OSPF services.

Find all chassis devices hosting OSPF services

This query identifies all devices that are hosting OSPF services. These devices are serving as routers within an autonomous system (AS). Each device identified has an IP address and a separate OSPF router IP address.

Example

```
1]  SELECT     e.entityName Entity_Name,
2]             o.routerId OSPF_Router_ID
3]  FROM       entityData e
4]  INNER JOIN hostedService h ON h.hostingEntityId = e.entityId
5]  INNER JOIN ospfService o ON o.entityId = h.hostedEntityId;
```

Description

The table below describes this query.

Table 301. Description of the query

Line numbers	Description
1-2	Specify the data to show in the results, as follows: • The IP address of the hosting entity, represented by `e.EntityName` • The ID of the hosted service, represented by `o.routerID`
3	Use the `entityData` table as the driving table for this query.
4	Restrict the entities returned to devices that host services. Do this by joining the `hostedService` table.
5	For each of the entities identified as devices hosting services, retrieve the OSPF service hosted on that device. Do this by joining the `ospfService` table to the query.

Results

The table below shows the results of this query.

Table 302. Results of the query

Entity name	OSPF router ID
172.18.1.2	22.130.159.0
172.18.2.4	22.130.53.0

Table 302. Results of the query (continued)	
Entity name	**OSPF router ID**
router1.ibm.net	172.20.4.16

Related concepts

Hosted services

A hosted service is a service or application running on a specific device. For example, a device can host BGP or OSPF services. NCIM can also model the fact that a software application, is running on a workstation.

Queries for collection information

These queries extract data on logical collections of devices.

Device collections are logical collections of devices. Examples of logical collections defined within NCIM include MPLS VPNs, global VLANs, and subnets. NCIM can also model OSPF areas.

Show all PIM adjacencies

This query returns details of all Protocol Independent Multicast (PIM) adjacencies.

Example

```
1]  SELECT     eA.entityName A, eZ.entityName Z
2]  FROM       topologyLinks t
3]  INNER JOIN connects c ON t.connectionId=c.connectionId
4]  INNER JOIN entityData eA ON eA.entityId=c.aEndEntityId
5]  INNER JOIN entityData eZ ON eZ.entityId=c.zEndEntityId
6]  INNER JOIN entityData et ON et.entityId = t.entityId
7]  WHERE et.entityName='PIMTopology';
```

Show PIM adjacencies for a device

This query shows Protocol Independent Multicast (PIM) adjacencies for a particular device.

Example

This example shows PIM adjacencies for the device 172.20.1.7.

```
1]  SELECT     eA.entityName A, eZ.entityName Z
2]  FROM       topologyLinks t
3]  INNER JOIN connects c ON t.connectionId=c.connectionId
4]  INNER JOIN entityData eA ON eA.entityId=c.aEndEntityId
5]  INNER JOIN entityData eZ ON eZ.entityId=c.zEndEntityId
6]  INNER JOIN entityData et ON et.entityId = t.entityId
7]  INNER JOIN entityData eAMain ON eAMain.entityId=eA.mainNodeEntityId
8]  INNER JOIN entityData eZMain ON eZMain.entityId=eZ.mainNodeEntityId
9]  WHERE et.entityName='PIMTopology'
10] and eAMain.entityName = '172.20.1.7' or eZMain.entityName = '172.20.1.7';
```

Find PIM enabled routers

This query returns a list of all routers that are enabled to use Protocol Independent Multicast (PIM).

Example

```
1]  SELECT     e.entityName,
2]             c.sysName,
3]             p.joinPruneInterval
4]  FROM       pimService p
5]  INNER JOIN hostedService h ON h.hostedEntityId=p.entityId
```

```
6]  INNER JOIN entityData e ON e.entityId=h.hostingEntityId
7]  INNER JOIN physicalChassis c ON c.entityId = e.mainNodeEntityId;
```

Find all devices in each subnet

This query identifies all of the subnets listed in the database. For each subnet the query provides the netmask of that subnet and the list of IP addresses collected within that subnet. The IP address collected within a subnet might refer to main nodes or interfaces; typically, they refer to interfaces.

Example

```
1]  SELECT     s.network Network,
2]             s.netmask Netmask,
3]             e.entityName Entity_Name
4]  FROM       subnet s
5]  INNER JOIN collects c ON c.collectingEntityId = s.entityId
6]  INNER JOIN entityData e ON e.entityId = c.collectedEntityId
7]  ORDER BY   s.network
```

Description

The table below describes this query.

Table 303. Description of the query

Line numbers	Description
1-3	Specify the data to show in the results, as follows: • The IP address of the collecting subnet, represented by `s.network` • The netmask of the subnet, represented by `s.netmask` • The name – usually an IP address – of an interface or main node within this subnet, represented by `e.entityName`
4	Use the `subnet` table as the driving table for this query. This enables the query to extract all the subnets in the database.
5	Retrieve a listing of all the collected entities within each subnet. At this point the collected entities are identified by their entity identifier only. The corresponding IP address is retrieved in the next line. Do this by joining the `collects` table.
6	Extract the entity data for each interface or main node collected within each subnet. Do this by joining the `entityData` table to the query. This enables the query to retrieve the IP address for each of the collected entities.
7	For readability purposes, order the results by the IP address of the collecting subnet.

Results

The table below shows the results of this query.

Table 304. Results of the query

Network	Netmask	Entity name
10.1.1.0	255.255.255.0	10.1.1.6
10.1.1.0	255.255.255.0	10.1.1.8
10.1.1.0	255.255.255.0	10.1.1.9
10.1.1.0	255.255.255.0	10.1.1.25
10.1.1.0	255.255.255.0	10.1.1.26

Table 304. Results of the query (continued)		
Network	**Netmask**	**Entity name**
10.1.1.0	255.255.255.0	10.1.1.27
172.18.1.0	255.255.255.0	172.18.1.30
172.18.1.0	255.255.255.0	172.18.1.31
172.20.11.0	255.255.255.248	172.20.11.54
172.20.11.0	255.255.255.248	172.20.11.75

Find all devices in a given VPN

This query identifies all of the VPNs listed in the database. For each VPN the query provides the name of that VPN and the list of IP addresses collected within that subnet. The IP address collected within a VPN might refer to main nodes or interfaces; typically they refer to interfaces.

Example

```
1]  SELECT     v.VPNName VPN_Name,
2]             v.VPNType VPN_Type,
3]             e.entityName Entity_Name
4]  FROM networkVpn v
5]  INNER JOIN collects c ON c.collectingEntityId = v.entityId
6]  INNER JOIN entityData e ON e.entityId = c.collectedEntityId
7]  ORDER BY   v.VPNName
```

Description

The table below describes this query.

Table 305. Description of the query	
Line numbers	**Description**
1-3	Specify the data to show in the results, as follows: • The IP address of the VPN, represented by `v.VPNName` • The VPN type, represented by `v.VPNType` • The name – usually an IP address – of an interface or main node within this VPN, represented by `e.entityName`
4	Use the `networkVpn` table as the driving table for this query. This enables the query to extract all the VPNs in the database.
5	Retrieve a listing of all the collected entities within each VPN. At this point the collected entities are identified by their entity identifier only. The corresponding IP address is retrieved in the next line. Do this by joining the `collects` table.
6	Extract the entity data for each interface or main node collected within each VPN. Do this by joining the `entityData` table to the query. This enables the query to retrieve the IP address for each of the collected entities
7	For readability purposes, order the results by the name of the collecting VPN.

Results

The table below shows the results of this query.

Table 306. Results of the query		
VPN name	**VPN type**	**Entity name**
VPN-BLUE	MPLS L2 PseudoWire	172.18.1.31
VPN-BLUE	MPLS L2 PseudoWire	172.20.11.54
VPN-BLUE	MPLS L2 PseudoWire	172.20.11.75
VPN-GREEN	MPLS L2 BGP VPN	10.1.1.26
VPN-GREEN	MPLS L2 BGP VPN	10.1.1.27
VPN-GREEN	MPLS L2 BGP VPN	172.18.1.30
VPN-PURPLE	MPLS IP VPN RT PAIR	10.1.1.59
VPN-PURPLE	MPLS IP VPN RT PAIR	10.1.1.75
VPN-RED	MPLS IP VPN	172.20.11.103
VPN-RED	MPLS IP VPN	172.20.11.111
VPN-WHITE	MPLS IP VPN MESH	172.18.1.233
VPN-WHITE	MPLS IP VPN MESH	172.18.1.240
VPN-YELLOW	MPLS IP VPN	10.1.1.6
VPN-YELLOW	MPLS IP VPN	10.1.1.8
VPN-YELLOW	MPLS IP VPN	10.1.1.9
VPN-YELLOW	MPLS IP VPN	10.1.1.25

Related concepts

Collection data

Collection data defines logical collections. Collections are defined in the `collects` table. Examples of logical collections defined within NCIM include MPLS VPNs, global VLANs, and subnets.

Queries for mapping and enumeration information

These sample queries extract mapping and enumeration data from NCIM.

Mappings and enumerations provide a means of looking up a database value in numerical or textual format and retrieving corresponding human-readable text.

Identify all the device hardware manufacturers listed in the database

This query provides a list of all device manufacturers held in the topology database.

The query uses the `mappings` table. This table provides lookups for alternative textual names. These lookups provide more human-readable text for fields. You can perform lookups in the `mappings` table for the types of information (or mapping groups) shown in the following table.

Table 307. Mapping groups supported by the mappings table		
Type of information	**String provided for lookup**	**Human-readable output of lookup**
MAC vendors	MAC address suffix information	Name of equipment vendor
Internet Assigned Number Authority (IANA) enterprise number	IANA enterprise number	Name of company with an enterprise section in the SNMP object MIB

Table 307. Mapping groups supported by the mappings table (continued)		
Type of information	**String provided for lookup**	**Human-readable output of lookup**
entPhysicalVendorType	MIB value for entPhysicalVendorType MIB variable	Vendor-specific hardware type of the physical entity
sysObjectId	MIB value for sysObjectId MIB variable	Vendor's authoritative identification of the network management subsystem contained in an entity

This query identifies the list of all device manufacturers held in the topology database by extracting a list of all mappings in the MACVendors mapping group in the mappings table.

The `mappings` table provides string-to-string mappings, whereas the `enumerations` table provides integer-to-string mappings.

Example

```
1]  SELECT    DISTINCT(mappingValue) Equipment_Vendor
2]  FROM      mappings m
3]  WHERE     mappingGroup = 'MACVendors'
4]  ORDER BY  mappingValue;
```

Description

The following table describes this query.

Table 308. Description of the query	
Line numbers	**Description**
1	Display the name of the equipment vendor. This is represented by `DISTINCT(mappingValue)`. Ensure that each name is listed only once by using the `DISTINCT` keyword.
2	Use the `mappings` table as the driving table for this query. This enables the query to extract all the mapping data in the database.
3	Limit the mappings to those that form part of the `MACVendors` mapping group.
4	Order the results by the name of the equipment vendor.

Results

The following table shows an example of the results of this query.

Table 309. Results of the query
Equipment vendor
360 Systems
3COM
3e Technologies International Inc.

Table 309. Results of the query (continued)
Equipment vendor
A-TREND TECHNOLOGY CO., LTD.
Abatron AG
ABB Automation Technology Products AB, Control
Abbey Systems Ltd
ABIT CORPORATION
AboveCable, Inc.
AbsoluteValue Systems, Inc.
AC Tech corporation DBA Advanced Digital
AC&T SYSTEM CO., LTD.
ACACIA NETWORKS, INC.

Related reference

Identify all the device hardware manufacturers listed in the database
This query provides a list of all device manufacturers held in the topology database.

Show all the entity types defined in the database

This query provides a list of entity types configured within the topology database. Entity type data includes a numerical key, a textual name for the entity type, and a category of entity to which the entity belongs.

Network Manager has the following entity categories:

- Element
- Collection
- Service
- Protocol endpoint
- Topology

This query uses the entityType table. This table contains every entity type in NCIM. If you want to define a new entity type, you need to update this table to include the entity type.

Example

```
1] SELECT    e.entityType Entity_Type,
2]          e.typeName Entity_Name,
3]          e.metaClass Category_of_Entity
4] FROM      entityType e;
```

Description

The following table describes this query.

Table 310. Description of the query	
Line numbers	**Description**
1-3	Specify the data to show in the results, as follows: • The numerical enumeration key value, represented by `e.entityType` • The corresponding human-readable string value, represented by `e.typeName` • Category of entity, represented by `e.metaClass`
4	All of this information is held in the `entityType` table.

Results

The following table shows some of the results of this query.

Table 311. Results of the query		
Entity type	**Entity name**	**Category of entity**
0	Unknown	Element
1	Chassis	Element
2	Interface	Element
3	Logical Interface	Element
4	localVlan	Element
5	Module	Element
6	PSU	Element
9	Fan	Element
10	Backplane	Element
11	Slot	Element
12	Sensor	Element

Related concepts

Topology data

When the network is discovered, both *core* NCIM tables and *entity attribute* tables are updated with topology data. These tables include Layer 1, Layer 2, Layer 3, device structure, routing protocol, containment, and technology-specific information.

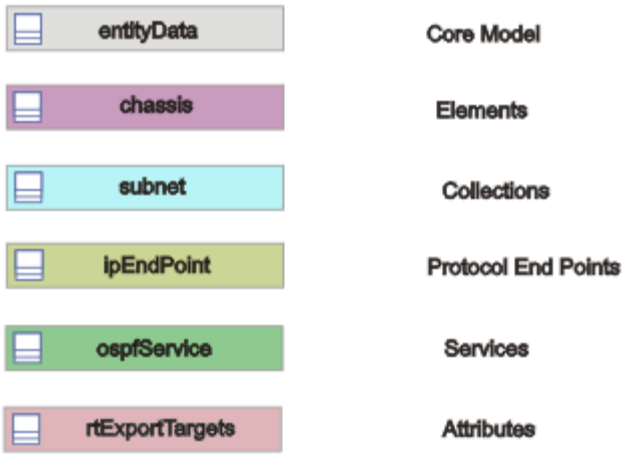
Chapter 22. NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

The NCIM database has the following schemas:

- Core schema
- Data schema

The NCIM database schemas are represented as a set of UML diagrams that model the relationships between topology data. Each class and relationship shown in the UML diagrams is modeled by a table in the NCIM relational database. The UML diagrams are color-coded and use the following color key.



Core schema

Use the following information to understand the NCIM database core schema.

The following UML diagram shows how NCIM models containment relationships.

In this diagram, the entity class has no connections to any of the other classes. This is intentional because the entity view is no longer part of the NCIM model as it got split into the entityData and domainMembers classes, and their corresponding tables. However, the entity class has been maintained as a database view partly for convenience as it makes some SQL easier to write but mainly to ensure backwards compatibility with previous versions of the schema. The entity class is shown in the diagram for completeness.

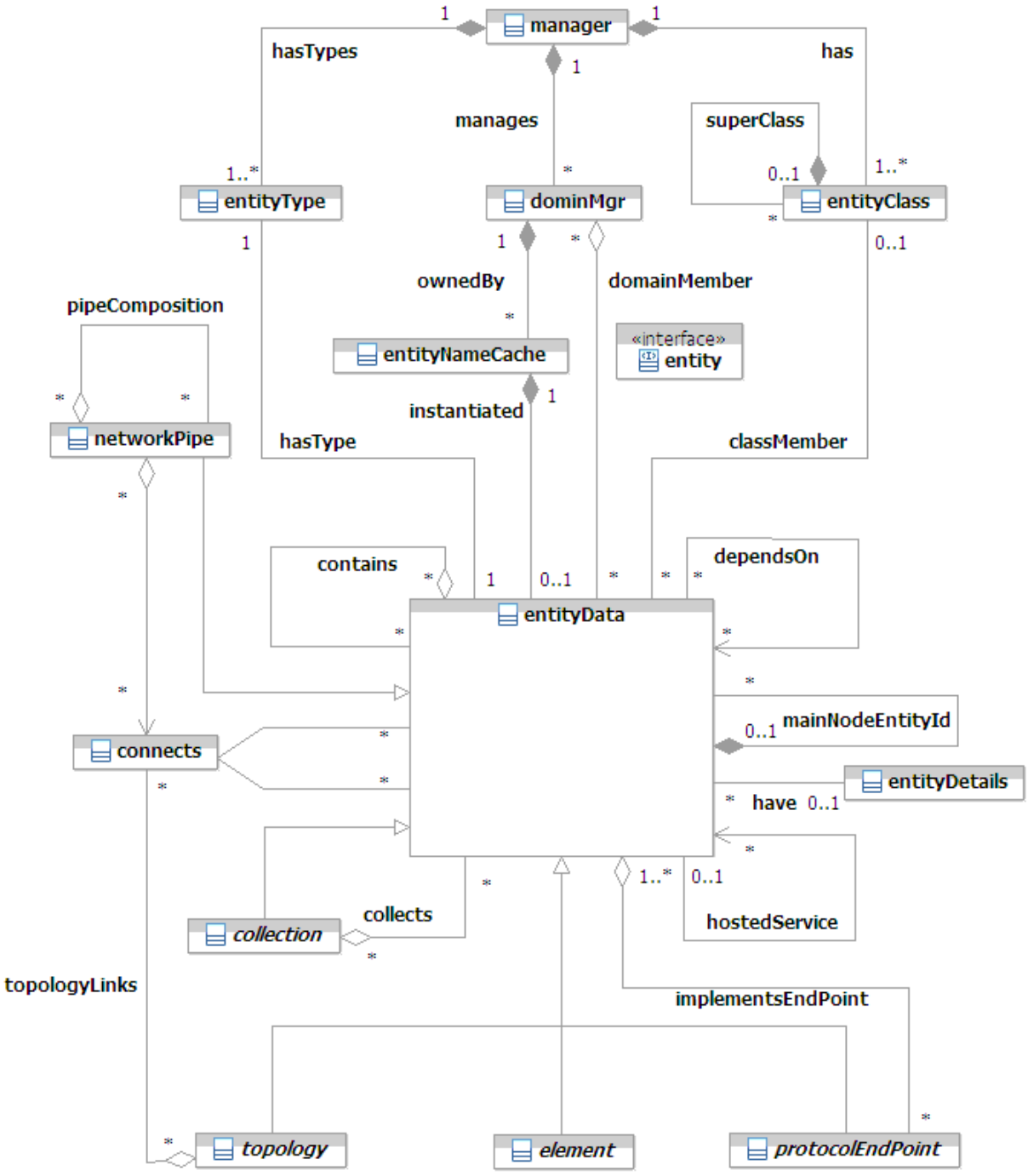


Figure 5. Core schema

Table 312 on page 556 describes the NCIM relationship database table and data dictionary that correspond to each class and relationship in the core schema.

Table 312. Classes and relationships for the core schema

NCIM table	Class or relationship	Related NCIM table or view	Data dictionary
Collection	Abstract Class	Not applicable	Not applicable

Table 312. Classes and relationships for the core schema (continued)

NCIM table	Class or relationship	Related NCIM table or view	Data dictionary
collects	Relationship	collects	“collects” on page 595
connects	Relationship	connects	“connects” on page 596
contains	Relationship	contains	“contains” on page 598
dependsOn	Relationship	entityDetails	“dependency” on page 599
domainMembers	Class	domainMembers	“domainMembers” on page 601
domainMgr	Class	domainMgr	“domainMgr” on page 602
Element	Abstract Class	NA	NA
entity	Class	entity	“entity” on page 620
entityClass	Class	entityClass	“entityClass” on page 604
entityData	Class	entityData	“entityData” on page 605
entityDetails	Class	entityDetails	“entityDetails” on page 607
entityNameCache	Class	entityNameCache	“entityNameCache” on page 607
entityType	Class	entityType	“entityType” on page 608
hostedService	Relationship	hostedService	“hostedService” on page 611
implementsEndPoint	Relationship	hostedService	“protocolEndPoint” on page 617
manager	Class	manager	“manager” on page 612
networkPipe	Class	networkPipe	“networkPipe” on page 613
pipeComposition	Class	pipeComposition	“pipeComposition” on page 614
protocolEndPoint	Class	hostedService	“protocolEndPoint” on page 617
topologyLinks	Relationship	hostedService	“topologyLinks” on page 618

Data schema

In the NCIM database, Network Manager topology data falls into different categories.

Related reference

dNCIM schema

The dNCIM database holds the containment model that is derived from the workingEntities.finalEntity, workingEntities.containment and layer tables, mainly fullTopology.entityByNeighbor. The model is built by the stitchers located in the dNCIM subdirectory, $NCHOME/precision/disco/stitchers/DNCIM. This is the version of the topology that is sent to the ncp_model component

BGP

Use this information to understand how the NCIM database models Border Gateway Protocol (BGP).

The following UML diagram shows how NCIM models BGP.

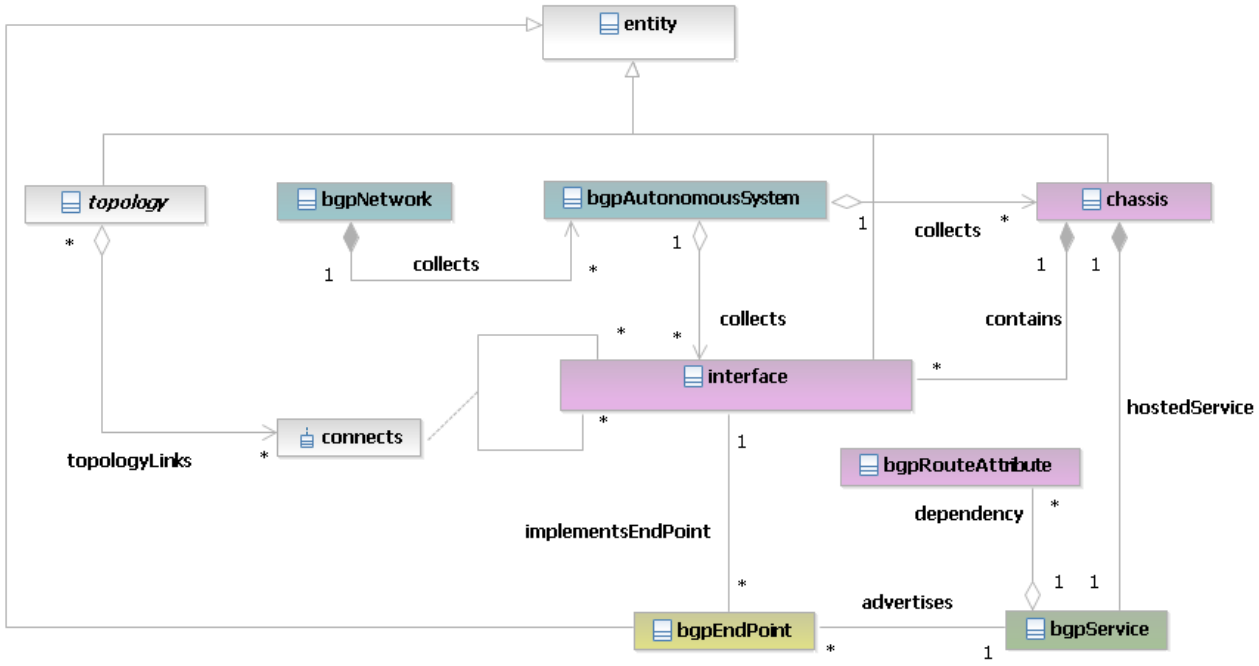


Figure 6. BGP schema

Table 313 on page 558 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model BGP.

Table 313. Classes and relationships for BGP

Item	Class or relationship	Data dictionary
bgpAutonomousSystem	Class	"bgpAutonomousSystem" on page 633
bgpEndPoint	Class	"bgpEndPoint" on page 634
bgpNetwork	Class	"bgpNetwork" on page 637
bgpRouteAttribute	Class	"bgpRouteAttribute" on page 637
bgpService	Class	"bgpService" on page 639

Table 313. Classes and relationships for BGP (continued)		
Item	**Class or relationship**	**Data dictionary**
chassis	Class	“physicalChassis” on page 710
collects	Relationship	“collects” on page 595
connects	Relationship	“connects” on page 596
contains	Relationship	“contains” on page 598
entity	Class	“entity” on page 620
hostedService	Relationship	“hostedService” on page 611
implementsEndPoint	Class	“protocolEndPoint” on page 617
interface	Class	“networkInterface” on page 687
topologyLinks	Relationship	“topologyLinks” on page 618

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Collections

Use this information to understand how the NCIM database models device collections, such as subnets, VPNs, and VLANs.

The following UML diagram shows how NCIM models device collections, such as subnets, VPNs and VLANs.

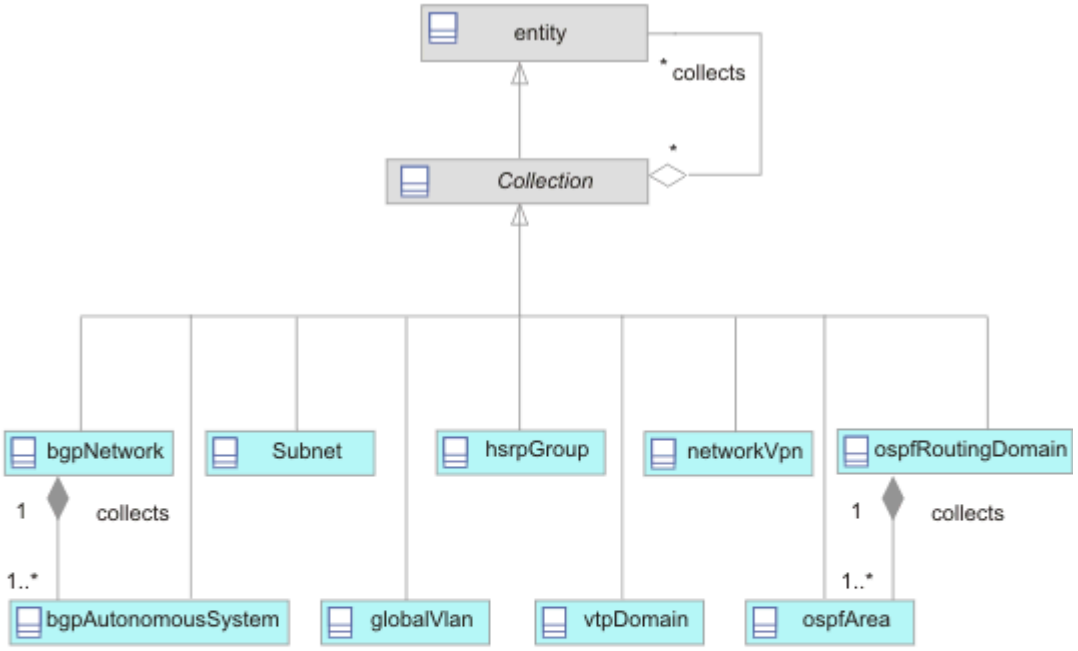


Figure 7. Device collections schema

Table 314 on page 560 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used for modelling device collections.

Table 314. Classes and relationships for device collections

Item	Class or relationship	Data dictionary
bgpAutonomousSystem	Class	"bgpAutonomousSystem" on page 633
bgpNetwork	Class	"bgpNetwork" on page 637
Collection	Abstract Class	Not applicable
collects	Relationship	"collects" on page 595
entity	Class	"entity" on page 620
globalVlan	Class	"globalVlan" on page 659
hsrpGroup	Class	"hsrpGroup" on page 662

Table 314. Classes and relationships for device collections (continued)		
Item	**Class or relationship**	**Data dictionary**
networkVpn	Class	"networkVpn" on page 691
ospfArea	Class	"ospfArea" on page 699
ospfRoutingDomain	Class	"ospfRoutingDomain" on page 701
pimNetwork	Class	"Collections" on page 559
VTPDomain subnet	Class Class	"vtpDomain" on page 750 "subnet" on page 746

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Containment

Use this information to learn how the NCIM database models containment relationships.

The following UML diagram shows how NCIM models containment relationships.

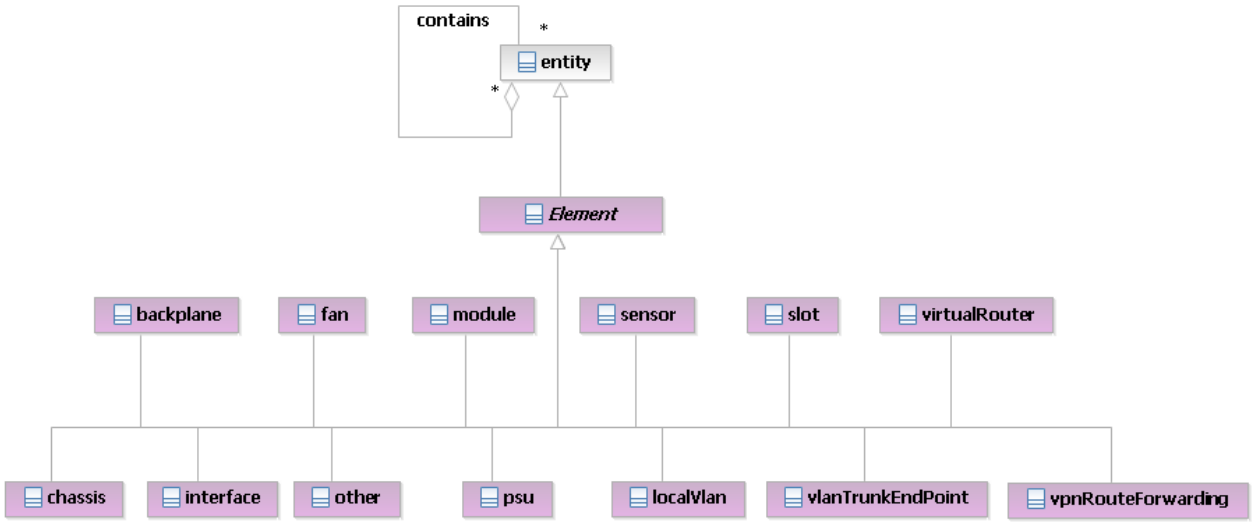


Figure 8. Containment schema

Table 315 on page 561 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model containment relationships.

Table 315. Classes and relationships for containment		
Item	**Class or relationship**	**Data dictionary**
backplane	Class	"backplane" on page 754
chassis	Class	"physicalChassis" on page 710
Element	Abstract Class	Not applicable

Table 315. Classes and relationships for containment (continued)		
Item	**Class or relationship**	**Data dictionary**
entity	Class	“entity” on page 620
fan	Class	“physicalFan” on page 716
interface	Class	“networkInterface” on page 687
localVlan	Class	“localVlan” on page 676
module	Class	“physicalCard” on page 706
other	Class	“physicalOther” on page 718
psu	Class	“physicalPowerSupply” on page 719
sensor	Class	“physicalSensor” on page 721
slot	Class	“physicalSlot” on page 724
vlanTrunkEndPoint	Class	“vlanTrunkEndPoint” on page 748
vpnRouteForwarding	Class	“vpnRouteForwarding” on page 749

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Endpoints

Use this information to understand how the NCIM database models endpoints.

The following UML diagram shows how NCIM models protocol endpoints. Not all endpoints are shown in the diagram; see the following table for a full list of endpoints.

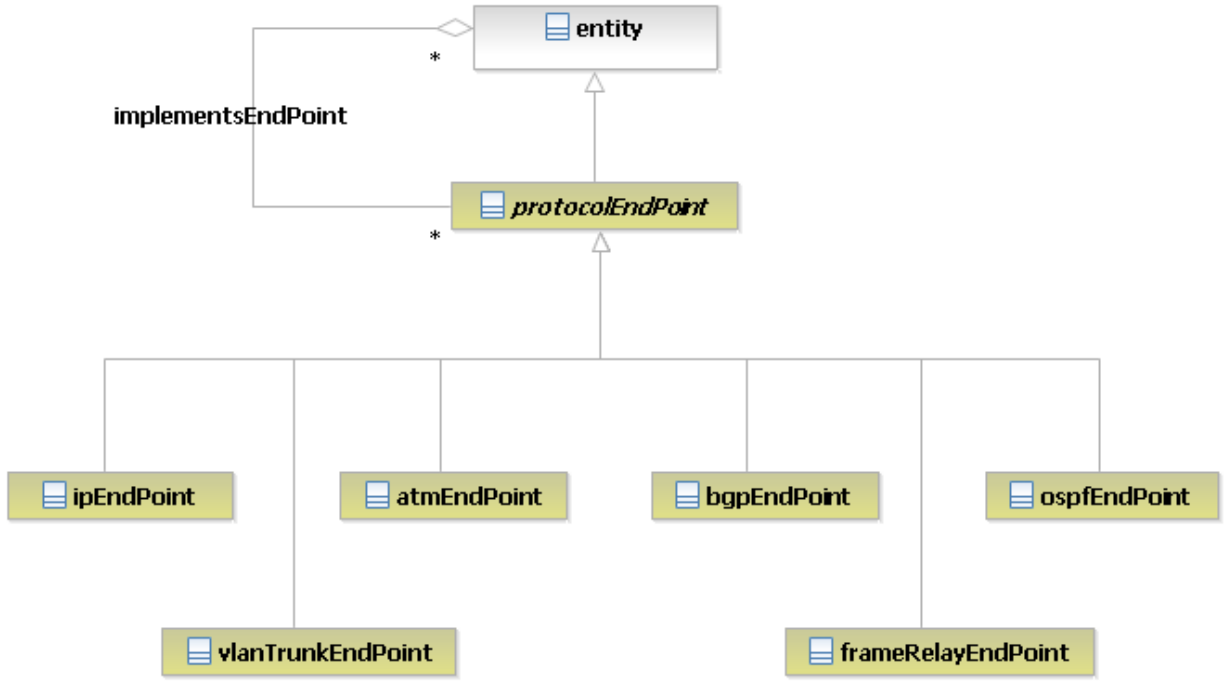


Figure 9. Protocol endpoints schema

Table 316 on page 563 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model endpoints.

Table 316. Classes and relationships for protocol endpoints

Item	Class or relationship	Data dictionary
atmEndPoint	Class	"atmEndPoint" on page 632
bgpEndPoint	Class	"bgpEndPoint" on page 634
entity	Class	"entity" on page 620
frameRelayEndPoint	Class	"frameRelayEndPoint" on page 656
igmpEndPoint	Class	"igmpEndPoint" on page 664
implementsEndPoint	Class	"protocolEndPoint" on page 617
ipEndPoint	Class	"ipEndPoint" on page 666
ipMRouteEndPoint	Class	"ipMRouteEndPoint" on page 669
mplsTETunnelEndPoint	Class	"mplsTETunnelEndPoint" on page 685
pimEndPoint	Class	"pimEndpoint" on page 726
portEndPoint	Class	"portEndPoint" on page 728

Table 316. Classes and relationships for protocol endpoints (continued)		
Item	**Class or relationship**	**Data dictionary**
protocolEndPoint	Class	"protocolEndPoint" on page 617
ospfEndPoint	Class	"ospfEndPoint" on page 700
vlanTrunkEndPoint	Class	"vlanTrunkEndPoint" on page 748
vpwsEndPoint	Class	"vpwsEndPoint" on page 749

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Geographical location

The NCIM database models geographical locations using several database tables.

The following UML diagram shows how NCIM models geographical locations.

Figure 10. Geographical location schema

Table 317 on page 564 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model geographical locations.

Table 317. Classes and relationships for geographical locations		
NCIM table	**Class or relationship**	**Data dictionary**
collects	Relationship	"collects" on page 595
entityData	Class	"entityData" on page 605
geographicLocation	Class	"geographicLocation" on page 657
geographicRegion	Class	"geographicRegion" on page 659

IP endpoints

Use this information to understand how the NCIM database models Internet Protocol (IP) endpoints.

The following UML diagram shows how NCIM models IP endpoints.

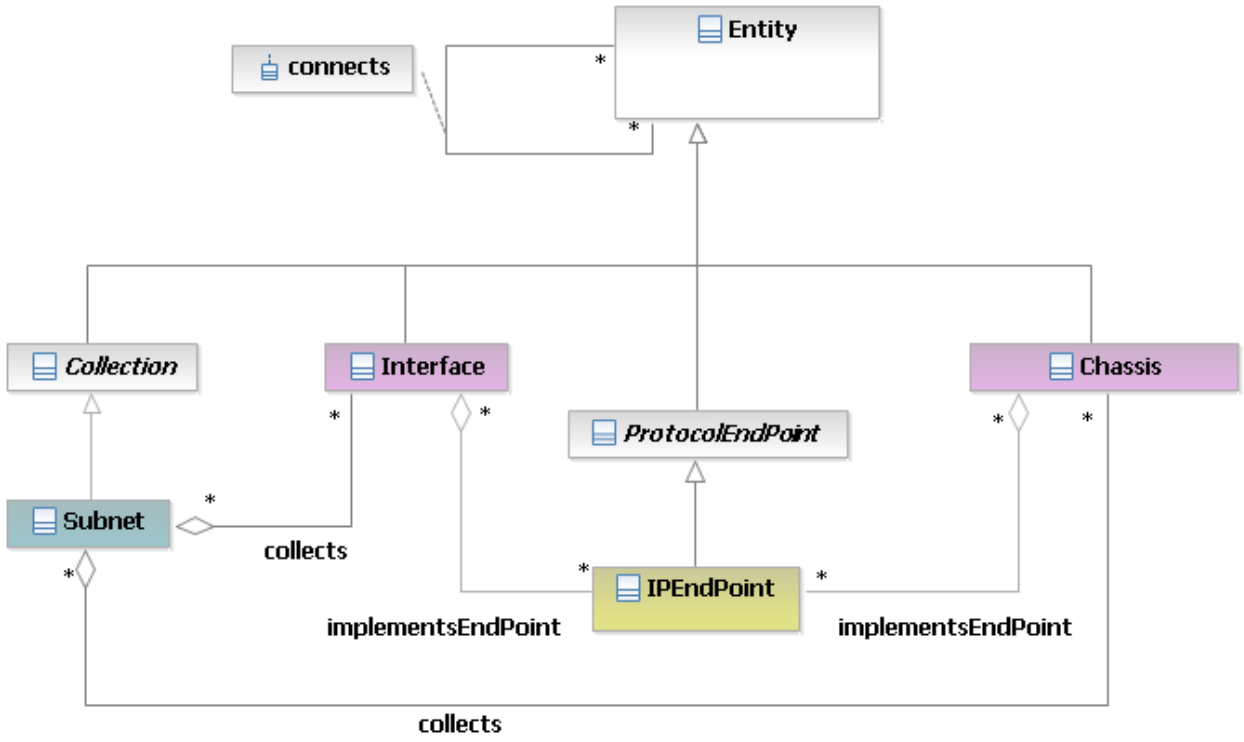


Figure 11. IP endpoints schema

Table 318 on page 565 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model IP endpoints.

Table 318. Classes and relationships for IP

Item	Class or relationship	Data dictionary
chassis	Class	"physicalChassis" on page 710
Collection	Abstract class	Not applicable
collects	Relationship	"collects" on page 595
connects	Relationship	"connects" on page 596
entity	Class	"entity" on page 620
implementsEndPoint	Class	"protocolEndPoint" on page 617
interface	Class	"networkInterface" on page 687
ipEndPoint	Class	"ipEndPoint" on page 666
protocolEndPoint	Class	"protocolEndPoint" on page 617

Table 318. Classes and relationships for IP (continued)		
Item	**Class or relationship**	**Data dictionary**
subnet	Class	“subnet” on page 746

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

LTE

In the NCIM database, Network Manager LTE topology data is modelled using a variety of NCIM tables.

LTE schema

Use the following information for a high-level view of the LTE schema.

The following UML diagram shows how NCIM models LTE entities and relationships.

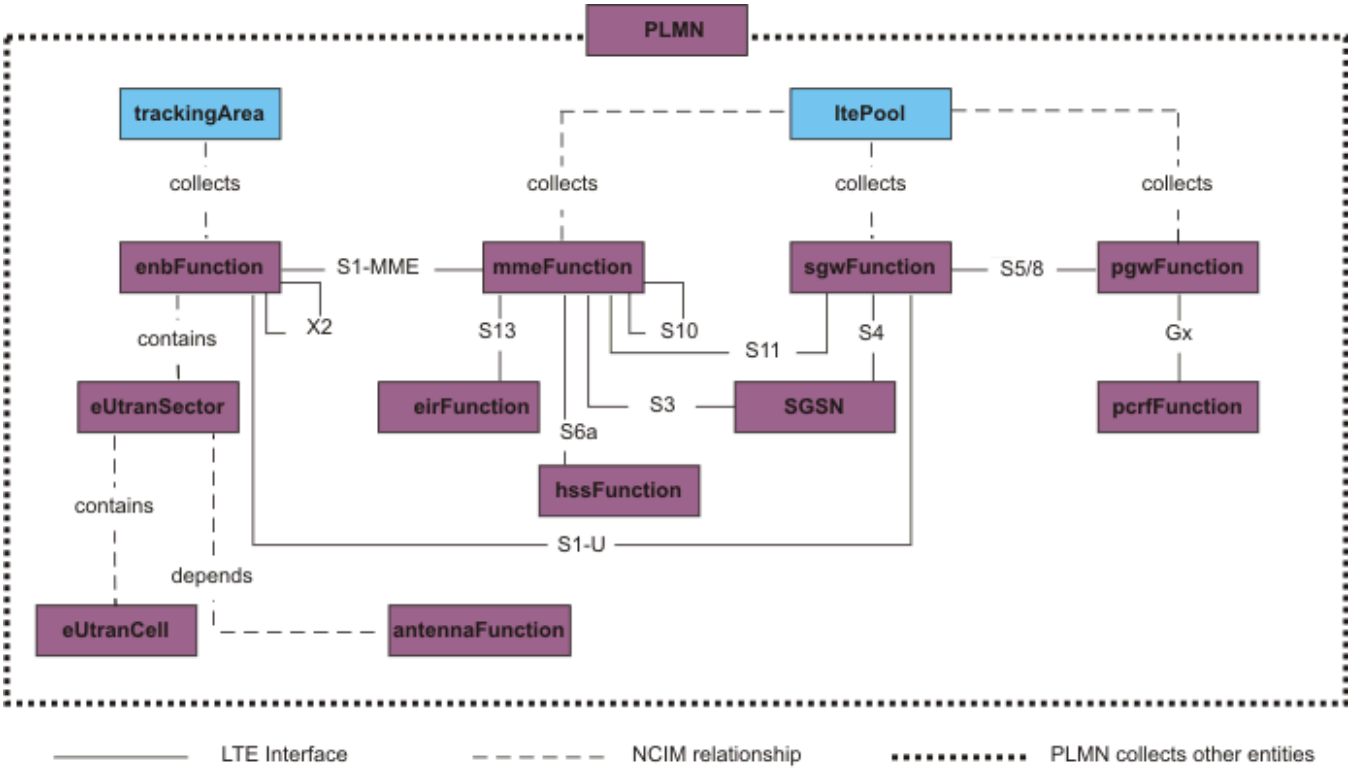


Figure 12. LTE schema

The following table describes the NCIM relationship database table that correspond to each class and relationship in the LTE schema.

Table 319. Classes and relationships for the LTE schema			
NCIM table	**Class or relationship**	**Related NCIM table or view**	**Data dictionary**
antennaFunction	Class	antennaFunction	“antennaFunction” on page 631
collects	Relationship	collects	“collects” on page 595

Table 319. Classes and relationships for the LTE schema (continued)			
NCIM table	**Class or relationship**	**Related NCIM table or view**	**Data dictionary**
contains	Relationship	contains	“contains” on page 598
depends	Relationship	depends	“dependency” on page 599
eirFunction	Class	eirFunction	“eirFunction” on page 649
enbFunction	Class	enbFunction	“enbFunction” on page 651
eUtranCell	Class	eUtranCell	“eUtranCell” on page 653
eUtranSector	Class	eUtranSector	“eUtranSector” on page 655
hssFunction	Class	hssFunction	“hssFunction” on page 662
ltePool	Class	ltePool	“ltePool” on page 679
mmeFunction	Class	mmeFunction	“mmeFunction” on page 681
pcrfFunction	Class	pcrfFunction	“pcrfFunction” on page 702
pgwFunction	Class	pgwFunction	“pgwFunction” on page 704
PLMN	Class	PLMN	“plmn” on page 728
sgwFunction	Class	sgwFunction	“sgwFunction” on page 744
SGSN	Class	ranSGSN	“ranSGSN” on page 742
trackingArea	Class	trackingArea	“trackingArea” on page 747

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

The following UML diagram shows how NCIM models the LTE interfaces.

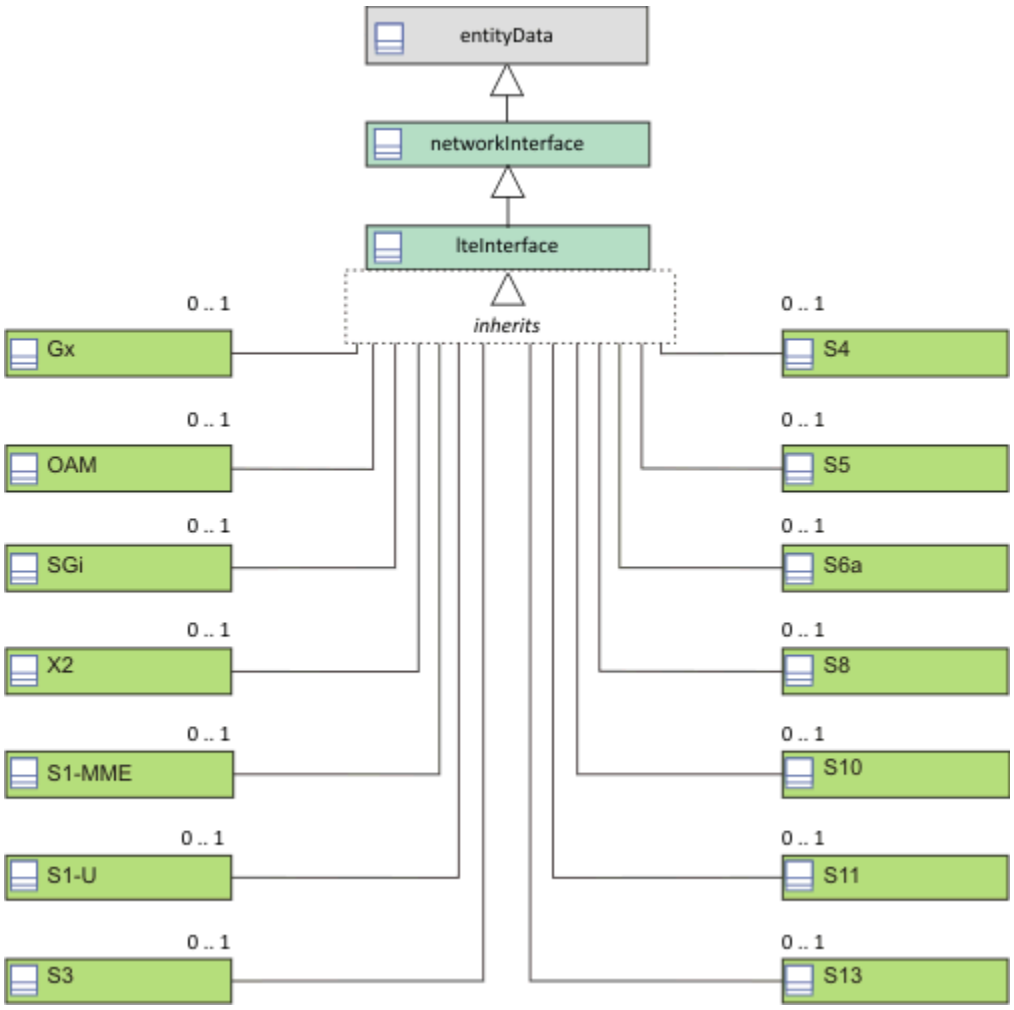


Figure 13. LTE interface schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model LTE interfaces.

Table 320. Classes and relationships for LTE interfaces

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
entityData	entityData	Class	“entityData” on page 605
lteInterface	lteInterface	Class	“lteInterface” on page 677
Gx	lteInterface	Class	“lteInterface” on page 677
networkInterface	networkInterface	Class	“networkInterface” on page 687
OAM	lteInterface	Class	“lteInterface” on page 677

Table 320. Classes and relationships for LTE interfaces (continued)			
UML element	**Modelled by NCIM table**	**Class or relationship**	**Data dictionary**
SGi	lteInterface	Class	“lteInterface” on page 677
X2	lteInterface	Class	“lteInterface” on page 677
S1-MME	lteInterface	Class	“lteInterface” on page 677
S1-U	lteInterface	Class	“lteInterface” on page 677
S3	lteInterface	Class	“lteInterface” on page 677
S4	lteInterface	Class	“lteInterface” on page 677
S5	lteInterface	Class	“lteInterface” on page 677
S6a	lteInterface	Class	“lteInterface” on page 677
S8	lteInterface	Class	“lteInterface” on page 677
S10	lteInterface	Class	“lteInterface” on page 677
S11	lteInterface	Class	“lteInterface” on page 677
S13	lteInterface	Class	“lteInterface” on page 677
X2	lteInterface	Class	“lteInterface” on page 677

LTE elements

In the NCIM database, Network Manager LTE elements are modelled using a variety of NCIM tables.

Note: Equipment Identity Register (EIR), Home Subscriber Server (HSS) and Policy and Charging Rules Function (PCRF) are not exclusively LTE elements. These three elements service a variety of technologies, including GSM, LTE, and UMTS. However, within the NCIM topology database, these elements are currently modelled only within LTE. They are therefore presented in the NCIM schema together with the other LTE elements.

Equipment Identity Register

The NCIM database models the Equipment Identity Register (EIR) using several database tables.

The following UML diagram shows how NCIM models the EIR.

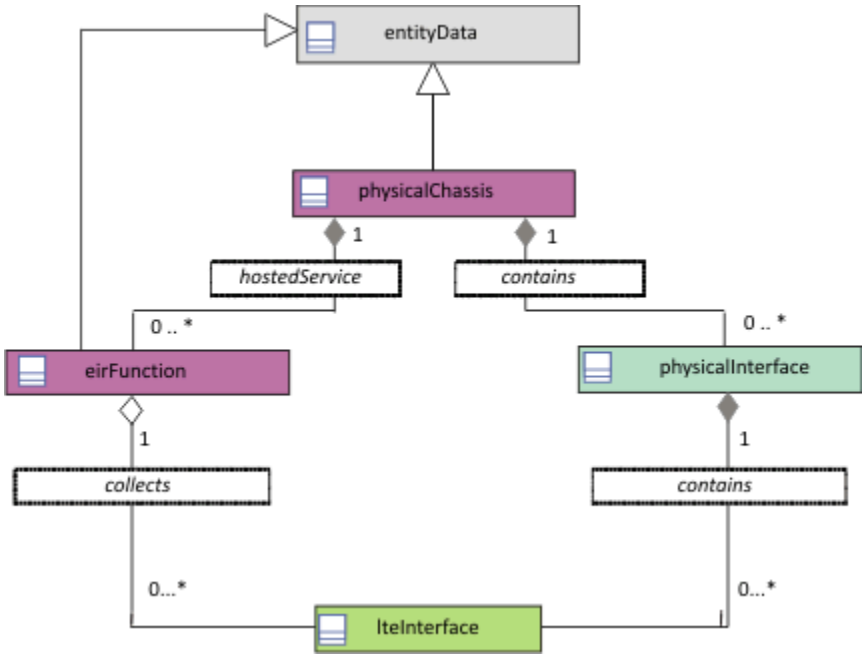


Figure 14. Equipment Identity Register (EIR) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model the EIR.

Table 321. Classes and relationships for EIR

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
collects	collects	Relationship	"connects" on page 596
contains	contains	Relationship	"contains" on page 598
eirFunction	eirFunction	Class	"eirFunction" on page 649
entityData	entityData	Class	"entityData" on page 605
lteInterface	lteInterface	Class	"lteInterface" on page 677
physicalChassis	physicalChassis	Class	"physicalChassis" on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	"networkInterface" on page 687

Related reference

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

Evolved NodeB

The NCIM database models the Evolved NodeB (eNodeB) using several database tables.

The following UML diagram shows how NCIM models the eNodeB.

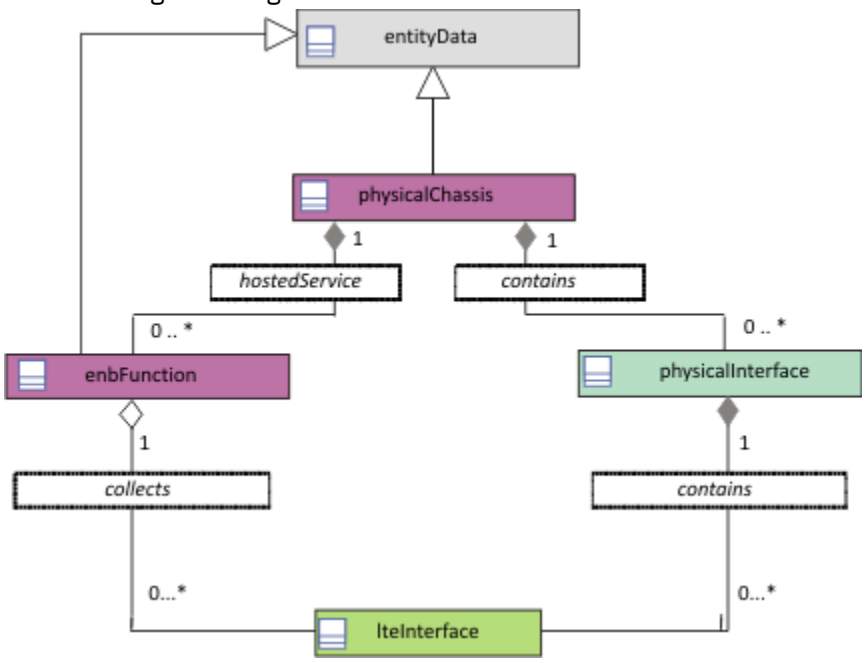


Figure 15. Evolved NodeB (eNodeB) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model the eNodeB.

Table 322. Classes and relationships for Evolved NodeB (eNodeB)

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
collects	collects	Relationship	"connects" on page 596
contains	contains	Relationship	"contains" on page 598
enbFunction	enbFunction	Class	"enbFunction" on page 651
entityData	entityData	Class	"entityData" on page 605
lteInterface	lteInterface	Class	"lteInterface" on page 677
physicalChassis	physicalChassis	Class	"physicalChassis" on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	"networkInterface" on page 687

Related reference

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

Home Subscriber Server

The NCIM database models the Home Subscriber Server (HSS) using several database tables.

The following UML diagram shows how NCIM models the HSS.

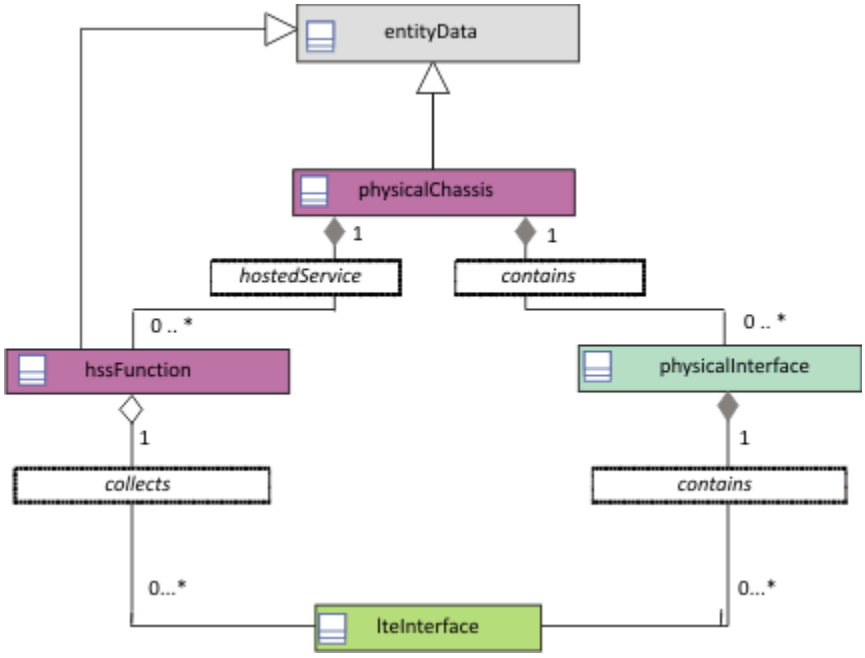


Figure 16. Home Subscriber Server (HSS) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model the Home Subscriber Server (HSS).

Table 323. Classes and relationships for Home Subscriber Server (HSS)

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
collects	collects	Relationship	"connects" on page 596
contains	contains	Relationship	"contains" on page 598
hssFunction	hssFunction	Class	"hssFunction" on page 662
entityData	entityData	Class	"entityData" on page 605
lteInterface	lteInterface	Class	"lteInterface" on page 677
physicalChassis	physicalChassis	Class	"physicalChassis" on page 710

Table 323. Classes and relationships for Home Subscriber Server (HSS) (continued)			
UML element	**Modelled by NCIM table**	**Class or relationship**	**Data dictionary**
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	“networkInterface” on page 687

Related reference

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

Mobility Management Entity

The NCIM database models the Mobility Management Entity (MME) using several database tables.

The following UML diagram shows how NCIM models the Mobility Management Entity.

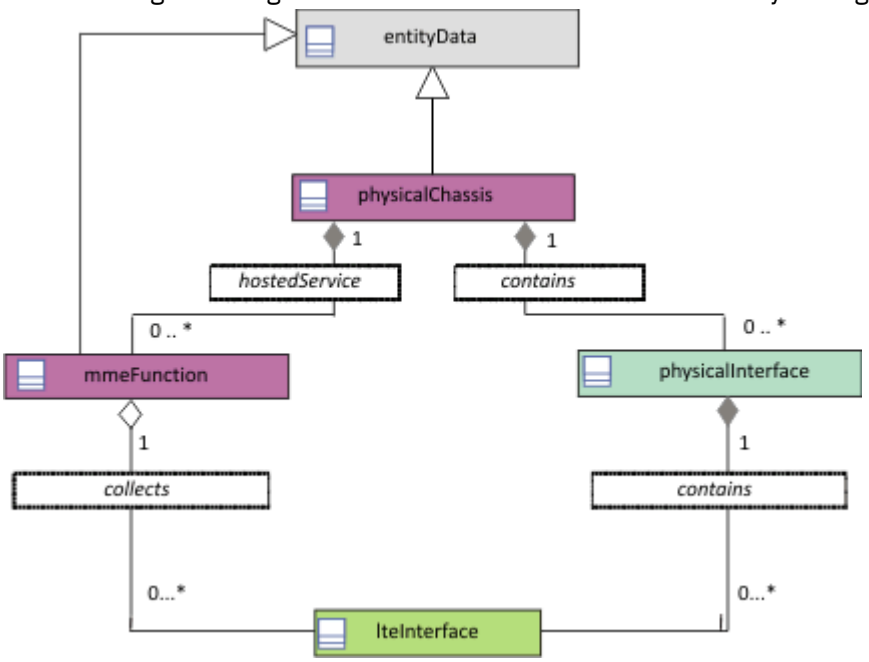


Figure 17. Mobility Management Entity (MME)

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model Mobility Management Entity.

Table 324. Classes and relationships for Mobility Management Entity (MME)			
UML element	**Modelled by NCIM table**	**Class or relationship**	**Data dictionary**
collects	collects	Relationship	“connects” on page 596
contains	contains	Relationship	“contains” on page 598
mmeFunction	mmeFunction	Class	“mmeFunction” on page 681

Table 324. Classes and relationships for Mobility Management Entity (MME) (continued)			
UML element	**Modelled by NCIM table**	**Class or relationship**	**Data dictionary**
entityData	entityData	Class	"entityData" on page 605
lteInterface	lteInterface	Class	"lteInterface" on page 677
physicalChassis	physicalChassis	Class	"physicalChassis" on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	"networkInterface" on page 687

Related reference

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

Packet Gateway

The NCIM database models the Packet Gateway (PGW) using several database tables.

The following UML diagram shows how NCIM models the Packet Gateway.

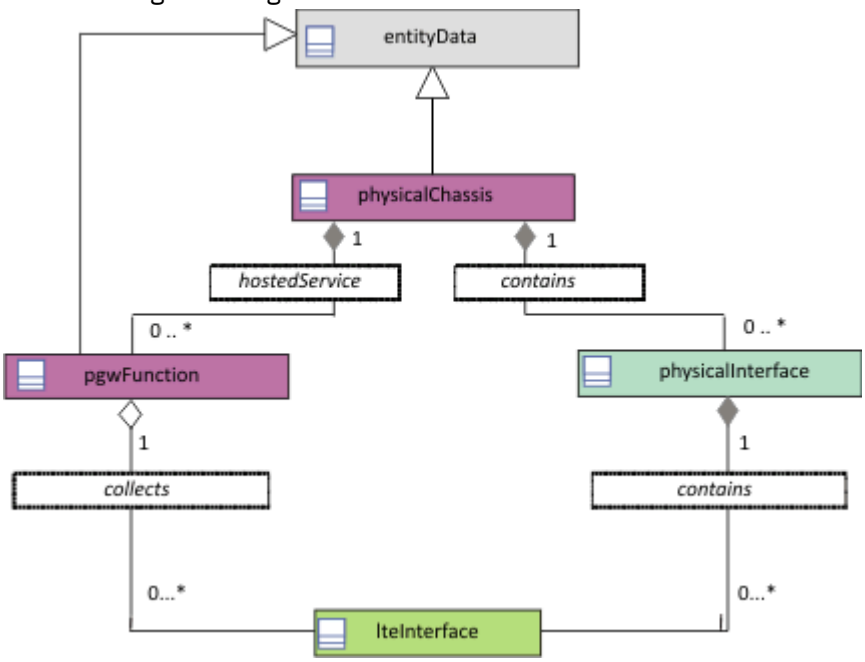


Figure 18. Packet Gateway (PGW) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model Packet Gateway.

Table 325. Classes and relationships for Packet Gateway (PGW)			
UML element	**Modelled by NCIM table**	**Class or relationship**	**Data dictionary**
collects	collects	Relationship	“connects” on page 596
contains	contains	Relationship	“contains” on page 598
pgwFunction	pgwFunction	Class	“pgwFunction” on page 704
entityData	entityData	Class	“entityData” on page 605
lteInterface	lteInterface	Class	“lteInterface” on page 677
physicalChassis	physicalChassis	Class	“physicalChassis” on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	“networkInterface” on page 687

Related reference

LTE interfaces

The NCIM database models LTE interfaces using several database tables.

Policy and Charging Rules Function

The NCIM database models the Policy and Charging Rules Function (PCRF) using several database tables.

The following UML diagram shows how NCIM models the Policy and Charging Rules Function (PCRF).

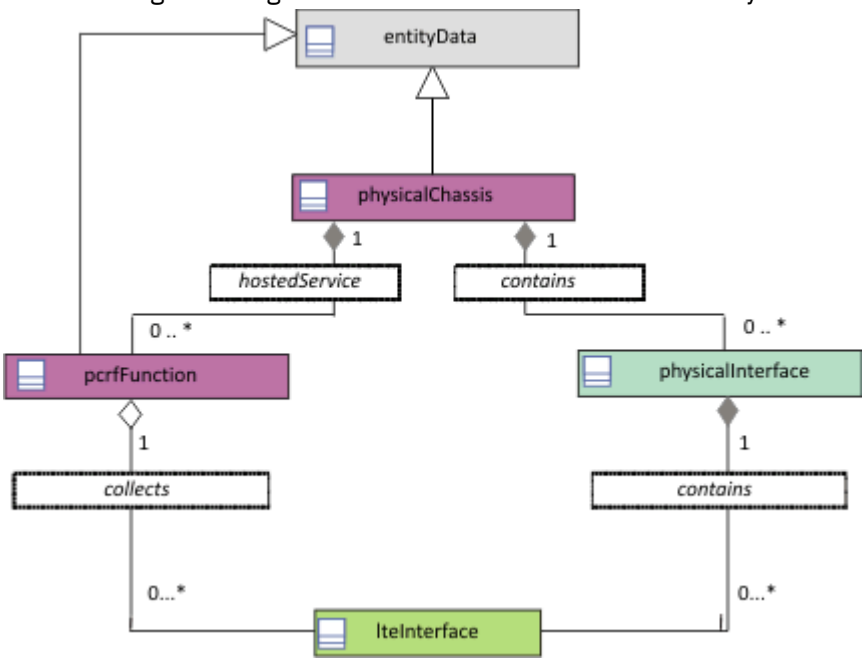


Figure 19. Policy and Charging Rules Function (PCRF) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model the Policy and Charging Rules Function (PCRF).

Table 326. Classes and relationships for Policy and Charging Rules Function (PCRF)

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
collects	collects	Relationship	“connects” on page 596
contains	contains	Relationship	“contains” on page 598
pcrfFunction	pcrfFunction	Class	“pcrfFunction” on page 702
entityData	entityData	Class	“entityData” on page 605
lteInterface	lteInterface	Class	“lteInterface” on page 677
physicalChassis	physicalChassis	Class	“physicalChassis” on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	“networkInterface” on page 687

Related reference

LTE interfaces
The NCIM database models LTE interfaces using several database tables.

Serving Gateway

The NCIM database models the Serving Gateway (SGW) using several database tables.

The following UML diagram shows how NCIM models the Serving Gateway.

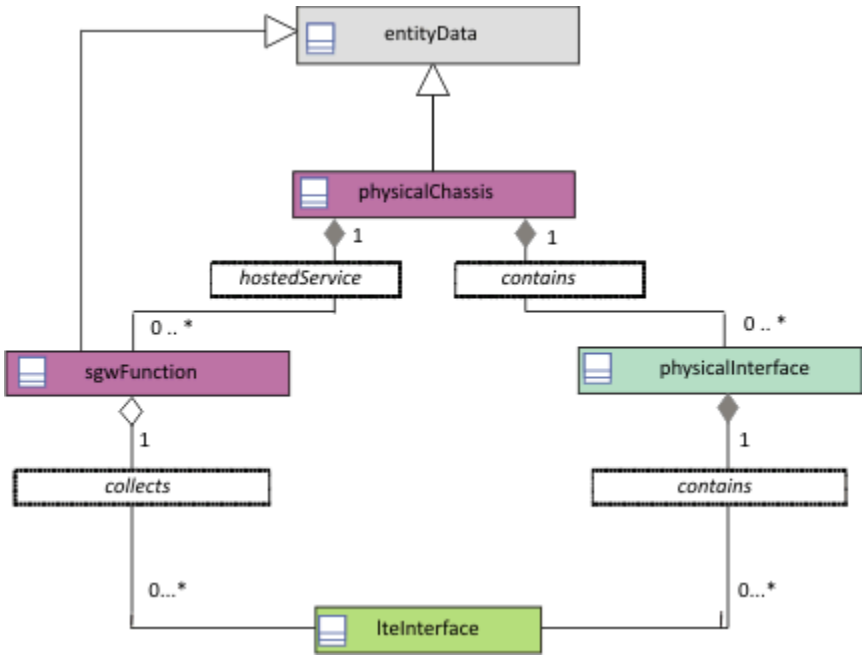


Figure 20. Serving Gateway (SGW) schema

The following table describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model Serving Gateway.

Table 327. Classes and relationships for Serving Gateway (SGW)

UML element	Modelled by NCIM table	Class or relationship	Data dictionary
collects	collects	Relationship	"connects" on page 596
contains	contains	Relationship	"contains" on page 598
sgwFunction	sgwFunction	Class	"sgwFunction" on page 744
entityData	entityData	Class	"entityData" on page 605
lteInterface	lteInterface	Class	"lteInterface" on page 677
physicalChassis	physicalChassis	Class	"physicalChassis" on page 710
physicalInterface	networkInterface Where the entity type of the interface has the value 2.	Class	"networkInterface" on page 687

Related reference
LTE interfaces

The NCIM database models LTE interfaces using several database tables.

MPLS traffic engineered (TE) tunnels

The NCIM database models MPLS TE tunnels using several databases.

The following UML diagram shows how NCIM models MPLS TE tunnels.

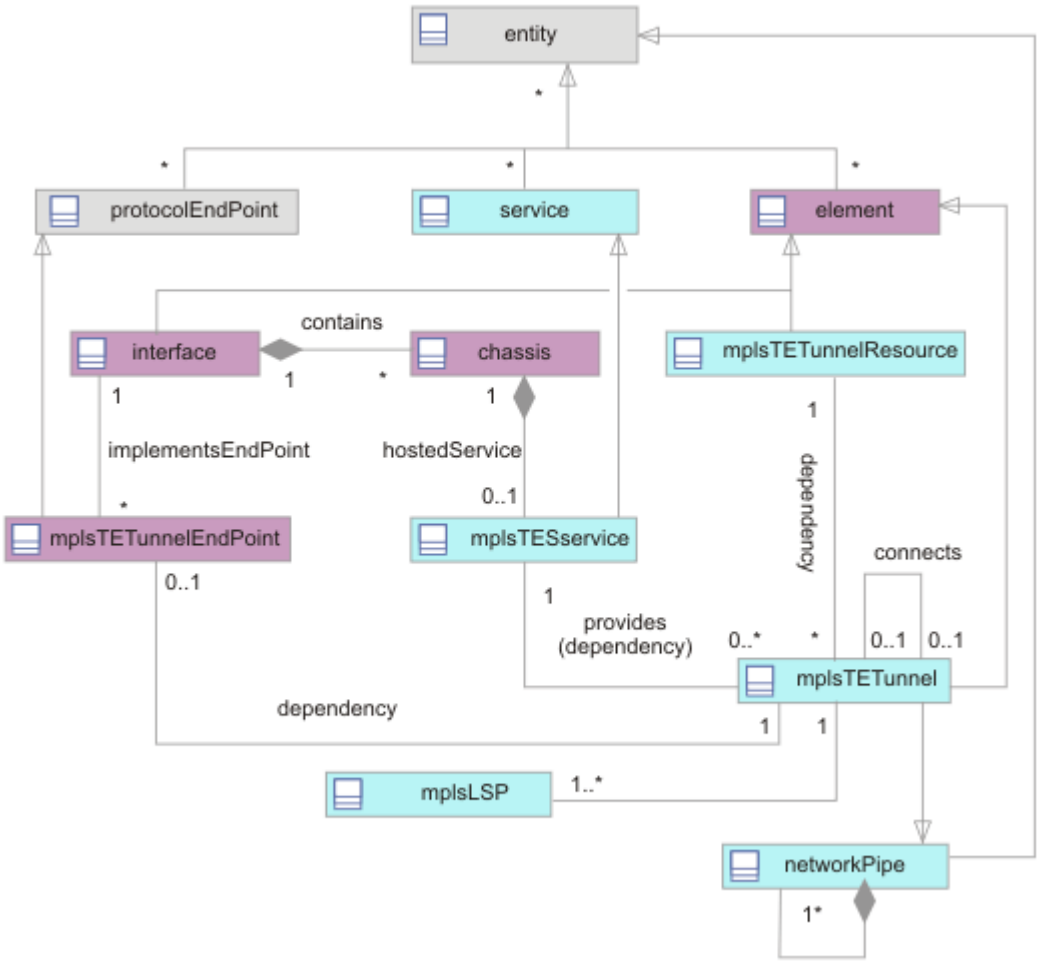


Figure 21. MPLS TE schema

Table 328 on page 578 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model MPLS TE tunnels.

Table 328. Classes and relationships for MPLS TE tunnels

NCIM table	Class or relationship	Data dictionary
chasssis	Class	"physicalChassis" on page 710
contains	Relationship	"contains" on page 598
dependency	Relationship	"dependency" on page 599
entity	Class	"entity" on page 620

Table 328. Classes and relationships for MPLS TE tunnels (continued)		
NCIM table	**Class or relationship**	**Data dictionary**
hostedService	Class	"hostedService" on page 611
interface	Class	"networkInterface" on page 687
mplsTEService	Class	"mplsTEService" on page 683
mplsTETunnel	Class	"mplsTETunnel" on page 683
mplsTETunnelEndPoint	Class	"mplsTETunnelEndPoint" on page 685
mplsTETunnelResource	Class	"mplsTETunnelResource" on page 685
mplsLSP	Class	"mplsLSP" on page 686

MPLS VPNs

Use this information to understand how the NCIM database models Multi Protocol Label Switching Virtual Private Networks (MPLS VPNs).

The following UML diagram shows how NCIM models MPLS VPNs.

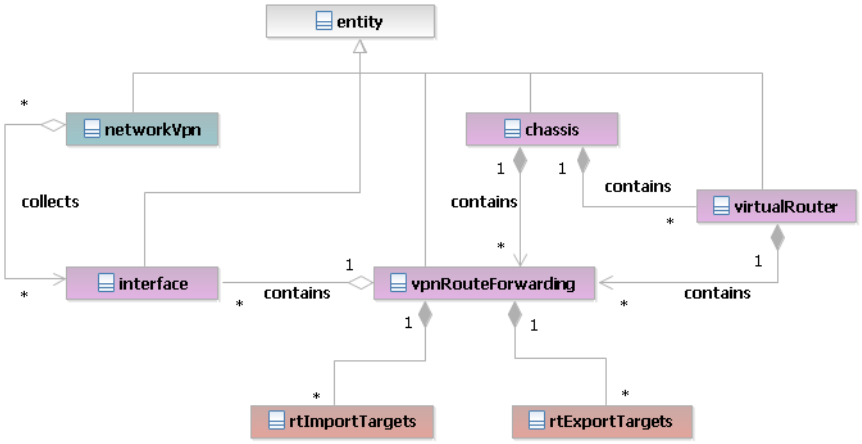


Figure 22. MPLS VPNs schema

Table 329 on page 579 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model MPLS VPNs.

Table 329. Classes and relationships for MPLS VPNs		
Item	**Class or relationship**	**Data dictionary**
chassis	Class	"physicalChassis" on page 710
collects	Relationship	"collects" on page 595
contains	Relationship	"contains" on page 598
entity	Class	"entity" on page 620

Table 329. Classes and relationships for MPLS VPNs (continued)		
Item	**Class or relationship**	**Data dictionary**
interface	Class	"networkInterface" on page 687
networkVpn	Class	"networkVpn" on page 691
RTExportTargets	Relationship	"rtExportList" on page 744
RTImportTargets	Relationship	"rtImportList" on page 744
vpnRouteForwarding	Class	"vpnRouteForwarding" on page 749

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

OSPF

Use this information to understand how the NCIM database models Open Shortest Path First (OSPF) protocols .

The following UML diagram shows how NCIM models OSPF.

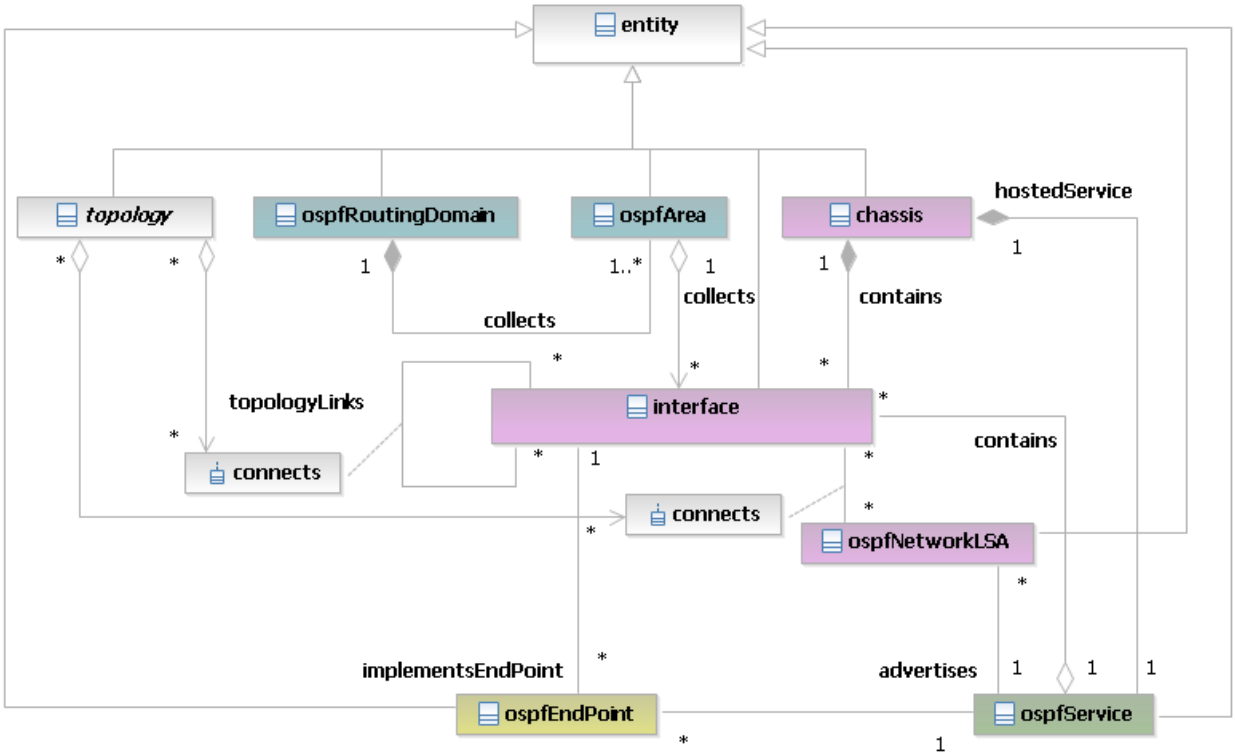


Figure 23. OSPF schema

Table 330 on page 581 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model OSPFs.

Table 330. Classes and relationships for OSPF		
Item	**Class or relationship**	**Data dictionary**
chassis	Class	“physicalChassis” on page 710
collects	Relationship	“collects” on page 595
connects	Relationship	“connects” on page 596
contains	Relationship	“contains” on page 598
entity	Class	“entity” on page 620
hostedService	Relationship	“hostedService” on page 611
implementsEndPoint	Class	“protocolEndPoint” on page 617
interface	Class	“networkInterface” on page 687
ospfArea	Class	“ospfArea” on page 699
ospfEndPoint	Class	“ospfEndPoint” on page 700
ospfNetworkLSA	Class	“ospfNetworkLSA” on page 701
ospfRoutingDomain	Class	“ospfRoutingDomain” on page 701
ospfService	Class	“ospfService” on page 701
topologyLinks	Relationship	“topologyLinks” on page 618

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Services

Use this information to understand how the NCIM database models the services that are offered by device interfaces, for example BGP services or OSPF services.

The following UML diagram shows how NCIM models services. Not all services are shown in the diagram; see the following table for a full list of services.

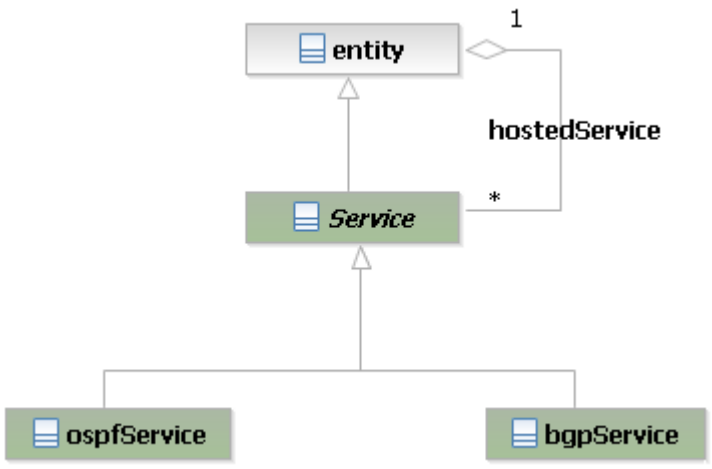


Figure 24. Services schema

Table 331 on page 582 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model services.

Table 331. Classes and relationships for services

Item	Class or relationship	Data dictionary
bgpService	Class	"bgpService" on page 639
entity	Class	"entity" on page 620
hostedService	Relationship	"hostedService" on page 611
igmpService	Class	"igmpService" on page 666
ipMRouteService	Class	"ipMRouteService" on page 671
itnmService	Class	"itnmService" on page 674
mplsTEService	Class	"mplsTEService" on page 683
ospfService	Class	"ospfService" on page 701
pimService	Class	"pimService" on page 727
Service	Abstract class	Not applicable

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

UMTS and GSM

In the NCIM database, Network Manager UMTS and GSM topology data is modelled using a variety of NCIM tables.

GSM

The NCIM database models GSM using several database tables.

The following UML diagram shows how NCIM models GSM.

Note: In addition to the standard relationships between entities shown in the core schema diagram, the GSM schema diagram models extra relationships between RAN entities. These RAN entity relationships have been added to make it easier to process RAN data. For example, the dependency relationship between `ranBaseStation` and `ranBaseStationController` was added to enable a single query to retrieve all RAN base stations managed by a given RAN base station controller.

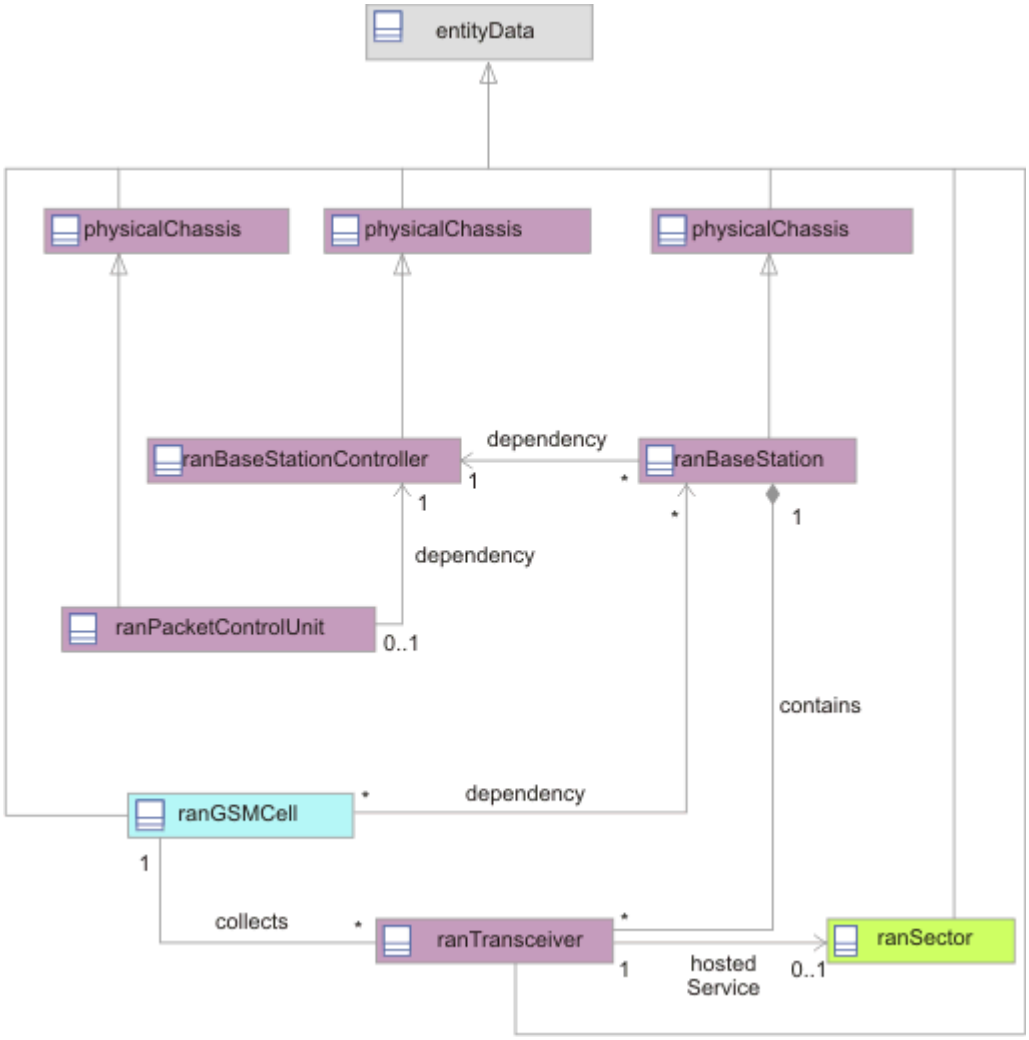


Figure 25. GSM schema

Table 332 on page 583 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model GSM.

Table 332. Classes and relationships for GSM

NCIM table	Class or relationship	Data dictionary
collects	Relationship	"collects" on page 595
connects	Relationship	"connects" on page 596
contains	Relationship	"contains" on page 598
dependency	Relationship	"dependency" on page 599
entityData	Class	"entityData" on page 605
physicalChasssis	Class	"physicalChassis" on page 710
ranBaseStationController	Class	"ranBaseStationController" on page 733

Table 332. Classes and relationships for GSM (continued)		
NCIM table	**Class or relationship**	**Data dictionary**
ranBaseStation	Class	“ranBaseStation” on page 732
ranGSMCell	Class	“ranGSMCell” on page 735
ranPacketControlUnit	Class	“ranPacketControlUnit” on page 739
ranSector	Class	“ranSector” on page 742
ranTransceiver	Class	“ranTransceiver” on page 743

RAN collections

The NCIM database models RAN collections using several database tables.

The following UML diagram shows how NCIM models RAN collections.

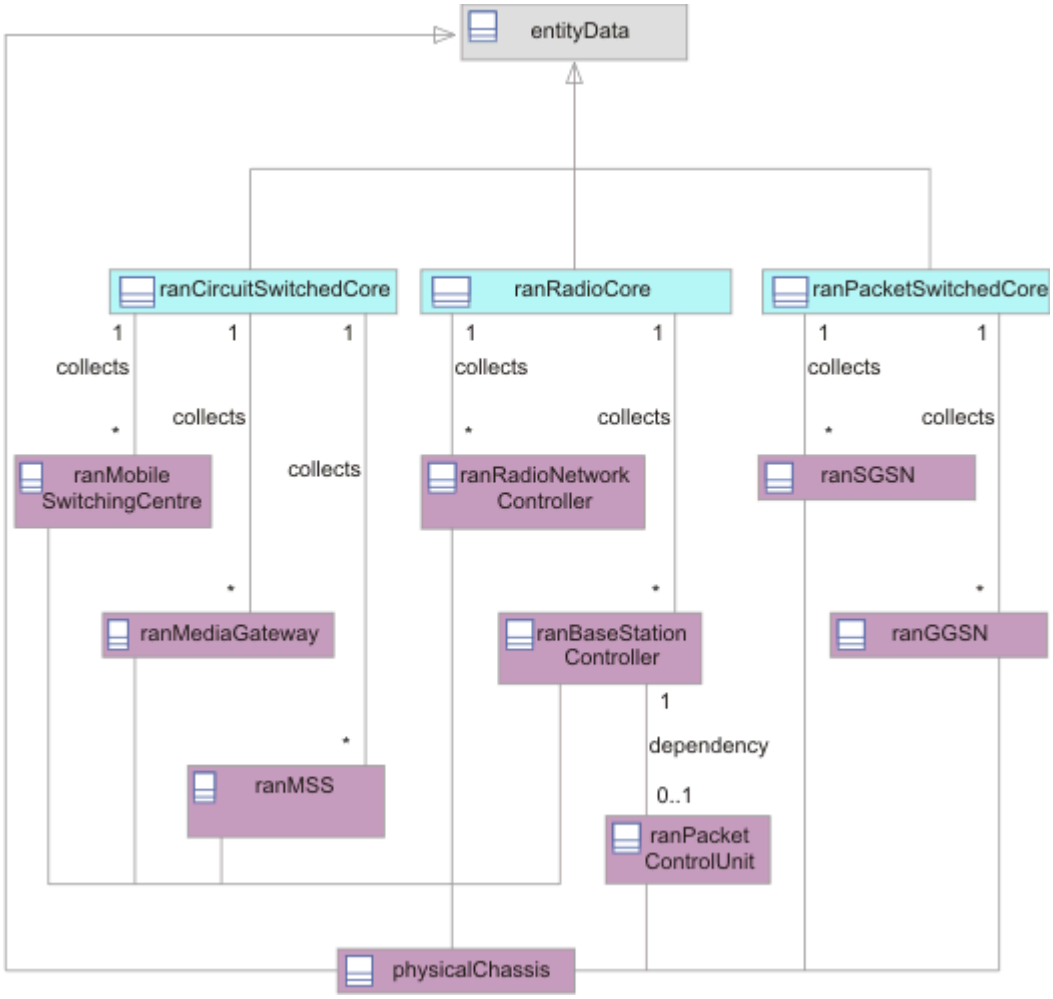


Figure 26. RAN collections

Table 333 on page 585 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model RAN collections.

Table 333. Classes and relationships for RAN collections		
NCIM table	**Class or relationship**	**Data dictionary**
collects	Relationship	“collects” on page 595
dependency	Relationship	“dependency” on page 599
entityData	Class	“entityData” on page 605
physicalChasssis	Class	“physicalChassis” on page 710
ranBaseStationController	Class	“ranBaseStationController” on page 733
ranCircuitSwitchedCore	Class	“ranCircuitSwitchedCore” on page 734
ranGGSN	Class	“ranGGSN” on page 734
ranMediaGateway	Class	“ranMediaGateway” on page 736
ranMSS	Class	“ranMSS” on page 737
ranMobileSwitchingCentre	Class	“ranMobileSwitchingCentre” on page 737
ranPacketControlUnit	Class	“ranPacketControlUnit” on page 739
ranPacketSwitchedCore	Class	“ranPacketSwitchedCore” on page 739
ranRadioCore	Class	“ranRadioCore” on page 740
ranRadioNetworkController	Class	“ranRadioNetworkController” on page 740
ranSGSN	Class	“ranSGSN” on page 742

RAN routing and location areas

The NCIM database models RAN routing and location areas using several database tables.

A routing area is the region within which an end user can move using the data service without having to update the Serving GPRS Serving Nodes (SGSN). The location area is the area within which an end user can move using the voice service without having to update the VLR (visitor location registrar, which indicates where you are physically located). Therefore RAN routing and location areas do not indicate geographic location but rather the responsibilities of the devices in the system.

The following UML diagram shows how NCIM models RAN routing and location areas.

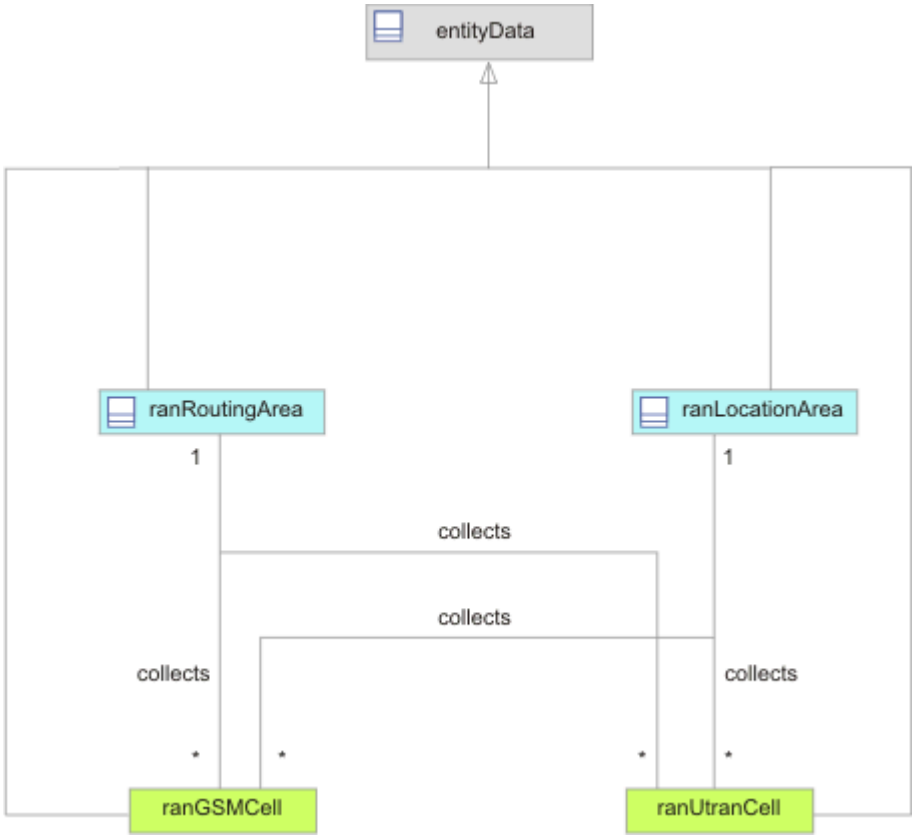


Figure 27. RAN location schema

Table 334 on page 586 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model radio access network entity location.

Table 334. Classes and relationships for radio access network entity location

NCIM table	Class or relationship	Data dictionary
collects	Relationship	"collects" on page 595
entityData	Class	"entityData" on page 605
ranGSMCell	Class	"ranGSMCell" on page 735
ranLocationArea	Class	"ranLocationArea" on page 736
ranRoutingArea	Class	"ranRoutingArea" on page 741
ranUtranCell	Class	"ranUtranCell" on page 743

UMTS

The NCIM database models UMTS using several database tables.

The following UML diagram shows how NCIM models UMTS.

Note: In addition to the standard relationships between entities shown in the core schema diagram, the UMTS schema diagram models extra relationships between RAN entities. These RAN entity relationships have been added to make it easier to process RAN data. For example, the dependency relationship between `ranUtranCell` and `ranNodeB` was added to enable a single query to retrieve all UTRAN cells managed by a given Node B entity.

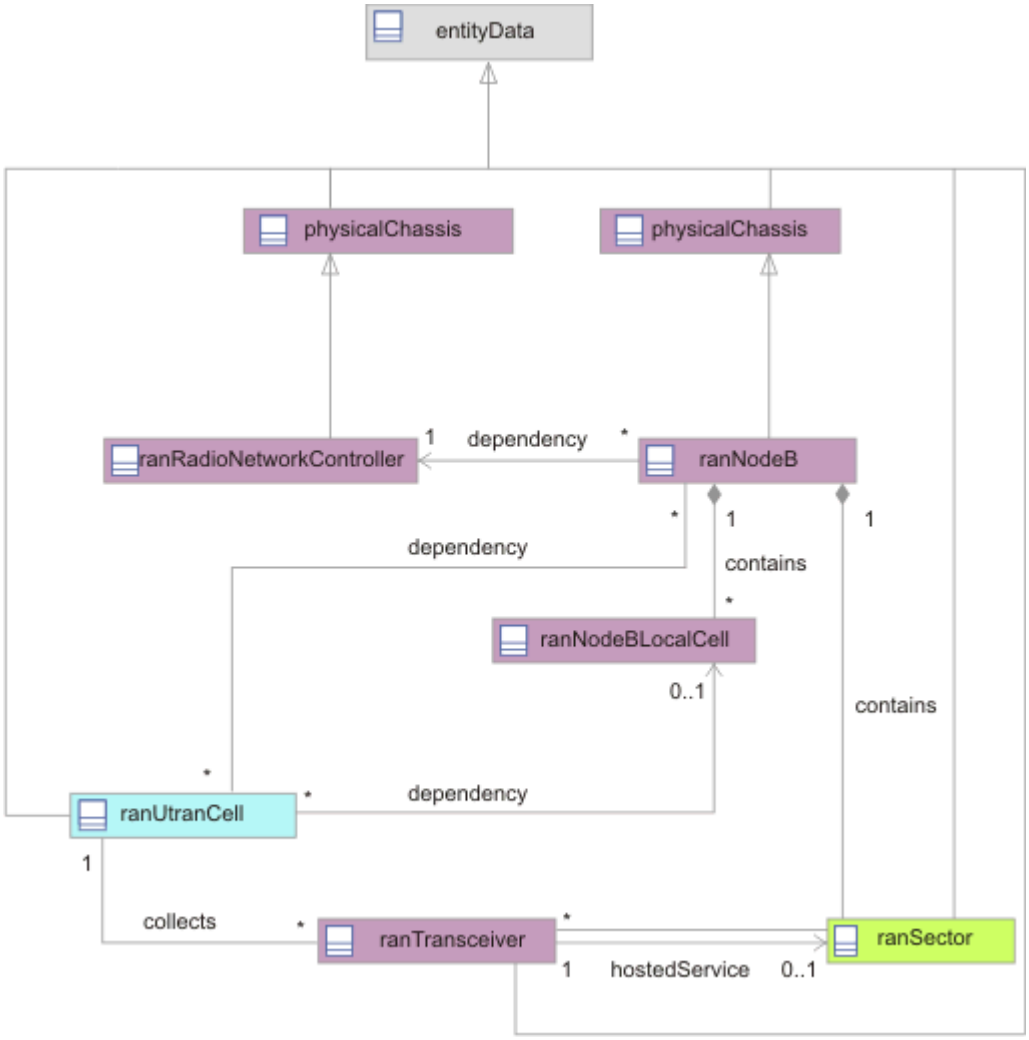


Figure 28. UMTS schema

Table 335 on page 587 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model UMTS.

Table 335. Classes and relationships for UMTS

NCIM table	Class or relationship	Data dictionary
collects	Relationship	“collects” on page 595
contains	Relationship	“contains” on page 598
dependency	Relationship	“dependency” on page 599
entityData	Class	“entityData” on page 605
hostedService	Relationship	“hostedService” on page 611
physicalChasssis	Class	“physicalChassis” on page 710
ranNodeB	Class	“ranNodeB” on page 738
ranNodeBLocalCell	Class	“ranNodeBLocalCell” on page 738

Table 335. Classes and relationships for UMTS (continued)		
NCIM table	**Class or relationship**	**Data dictionary**
ranRadioNetworkController	Class	“ranRadioNetworkController” on page 740
ranSector	Class	“ranSector” on page 742
ranTransceiver	Class	“ranTransceiver” on page 743
ranUtranCell	Class	“ranUtranCell” on page 743

VLANs

Use this information to understand how the NCIM database models virtual local area networks (VLANs).

The following UML diagram shows how NCIM models VLANs.

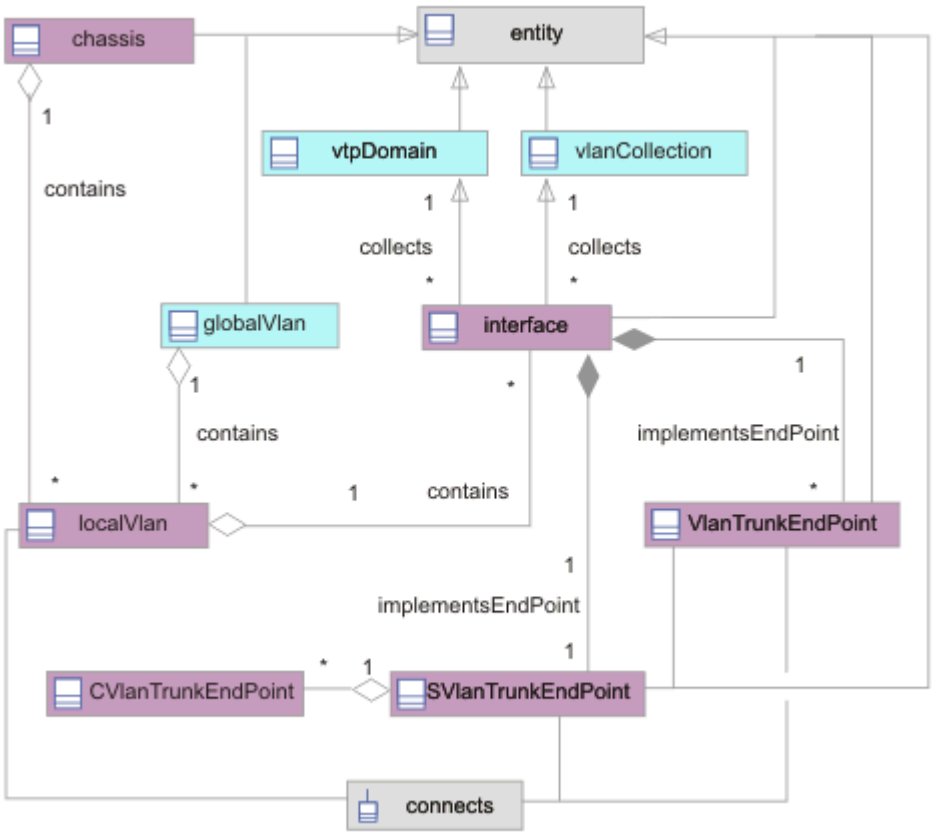


Figure 29. VLAN schema

Table 336 on page 589 describes the NCIM relationship database tables and data dictionary that correspond to each class and relationship used to model VLANs.

Table 336. Classes and relationships for VLANs		
Item	**Class or relationship**	**Data dictionary**
chassis	Class	“physicalChassis” on page 710
collects	Relationship	“collects” on page 595
contains	Relationship	“contains” on page 598
entity	Class	“entity” on page 620
globalVlan	Class	“globalVlan” on page 659
interface	Class	“networkInterface” on page 687
localVlan	Class	“localVlan” on page 676
implementsEndPoint	Class	“protocolEndPoint” on page 617
vlanCollection	Class	
vtpDomain	Class	“vtpDomain” on page 750
VLANTrunkEndPoint	Class	“vlanTrunkEndPoint” on page 748

Related concepts

NCIM topology database schemas

Use this information to understand how the relationships between topology data are modelled.

Chapter 23. Data dictionary

The NCIM topology database schema is made up of a set of relational database tables that represent the topology model.

Related concepts

Topology data

When the network is discovered, both *core* NCIM tables and *entity attribute* tables are updated with topology data. These tables include Layer 1, Layer 2, Layer 3, device structure, routing protocol, containment, and technology-specific information.

Related reference

dNCIM schema

The dNCIM database holds the containment model that is derived from the workingEntities.finalEntity, workingEntities.containment and layer tables, mainly fullTopology.entityByNeighbor. The model is built by the stitchers located in the dNCIM subdirectory, $NCHOME/precision/disco/stitchers/DNCIM. This is the version of the topology that is sent to the ncp_model component

Core tables

Core tables define entities and relationships between them. Core tables do not provide detailed attribute information on the entities.

Core tables include those tables that define the domains, for example the domainMgr table and the domainSummary table, and also the `entityData` table, which provides generic information about an entity, for example entityName and entityType.

The core table models the following categories of topology data:

Domains
A domain is a scoped set of entities that are discovered and managed by an application, such as Network Manager. Domains are represented using the `domainMgr` table. Membership of entities within these domains is represented using the `domainMembers` table.

Entities
An entity is a topology database concept. All devices and device components discovered by Network Manager are entities. Also device collections such as VPNs and VLANs, as well as pieces of topology that form a complex connection, are entities. Generic information for all entities within NCIM is held within the `entityData` table. The `entity` view joins data from the `entityData` and `domainMembers` tables and is equivalent to the `entity` table that existed in Network Manager versions 3.8 and earlier.

Containment relationships
Containment relationships express physical and logical containment. These relationships are represented by the `contains` table.

Connectivity relationships
Connectivity relationships are relationships between entities. These relationships are represented by the `connects` table. Other tables used to define connectivity include the networkPipe, pipeComposition, and topologyLinks tables.

Collection relationships
Collection relationships enable NCIM to model collections of entities, such as MPLS VPNs, global VLANs and subnets. The relationship is represented by the `collects` table.

Dependency relationships
Dependency relationships express a generic dependency relationship between two entities. This relationship is represented by the `dependency` table.

Mappings
Mappings provide a means of looking up a database value in numerical or textual format and retrieving corresponding human-readable text.

Related concepts

Topology data

When the network is discovered, both *core* NCIM tables and *entity attribute* tables are updated with topology data. These tables include Layer 1, Layer 2, Layer 3, device structure, routing protocol, containment, and technology-specific information.

aggregatedLink

The aggregatedLink table records the entity IDs of aggregated links in Link Aggregation Groups.

The following table describes the aggregatedLink table.

Table 337. aggregatedLink table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Primary key Not null	Automatically incremented ID that provides a unique value for each aggregated link across all domains.

aggregationDomain

The aggregationDomain table records the timestamps of when the last aggregation for an entity was done. If an entity is already up to date, it is not updated again.

The following table describes the aggregationDomain table.

Table 338. aggregationDomain table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Primary key Not null	Automatically incremented ID that provides a unique value for each entity across all domains.
domainMgrId	32-bit integer	Foreign key Primary key Not null	The identifier of the domain from the domainMgr table.
createTime		Not null	The time that the entity was created.
changeTime		Not null	The time that the entity was updated.

CIDRinfo

The CIDRinfo table provides the means to map between different representations of subnets and subnet masks. This table belongs to the category *mappings*.

The following table describes the CIDRinfo table.

Table 339. CIDRInfo table			
Column name	**Type**	**Constraints**	**Description**
maskBits	32-bit integer	Primary key Not null	The number of bits in the mask. For example, for a class C network, this is 24.
CIDRString	7-character string	Not null	The Classless Inter-Domain Routing (CIDR) notation for the subnet. For example, for a class C network, this is /24.
inverseMask	IP Address	Not null	The inverse mask for the network. The inverse mask acts as a wildcard for OSPF and ACLs.
numHosts	64-bit integer	Not null	The number of IP addresses in the network. For example, for a class C network this is 256.
numClassC	Double precision floating point number	Not null	The number of class C networks within the subnet.
netmask	15-character string	Not null	The subnet mask for the network. For example, for a class C network this is 255.255.255.0.

The following table summarizes the information in the CIDRInfo table.

Table 340. Summary of the information in the CIDRInfo table			
Subnet mask	**Bits**	**CIDR notation**	**Number of hosts**
0.0.0.0	0	/0	4294967296
128.0.0.0	1	/1	2147483648
192.0.0.0	2	/2	1073741824
224.0.0.0	3	/3	536870912
240.0.0.0	4	/4	268435456
248.0.0.0	5	/5	134217728
252.0.0.0	6	/6	67108864
254.0.0.0	7	/7	33554432
255.0.0.0	8	/8 (A)	16777216
255.128.0.0	9	/9	8388608
255.192.0.0	10	/10	4194304
255.224.0.0	11	/11	2097152
255.240.0.0	12	/12	1048576
255.248.0.0	13	/13	524288
255.252.0.0	14	/14	262144

Table 340. Summary of the information in the CIDRInfo table (continued)

Subnet mask	Bits	CIDR notation	Number of hosts
255.254.0.0	15	/15	131072
255.255.0.0	16	/16 (B)	65536
255.255.128.0	17	/17	32768
255.255.192.0	18	/18	16384
255.255.224.0	19	/19	8192
255.255.240.0	20	/20	4096
255.255.248.0	21	/21	2048
255.255.252.0	22	/22	1024
255.255.254.0	23	/23	512
255.255.255.0	24	/24 (C)	256
255.255.255.128	25	/25	128
255.255.255.192	26	/26	64
255.255.255.224	27	/27	32
255.255.255.240	28	/28	16
255.255.255.248	29	/29	8
255.255.255.252	30	/30	4
255.255.255.254	31	/31	2
255.255.255.255	32	/32	1

classMembers

The classMembers table specifies the device class to which a specific entity belongs. This table belongs to the category *entities*.

This table is populated only for entities that are chassis devices.

The following table describes the classMembers table.

Table 341. classMembers table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	Foreign key from the `entityData` table. Specifies the entity ID of the chassis device.
classId	32-bit integer	Foreign key Not null	Foreign key from the entityClass table. Specifies the ID of the class to which the chassis belongs.

collects

The `collects` table stores data on collections of entities, such as subnets and MPLS VPNs. This table belongs to the category *collections*.

The `sequence` column of the `collects` table allows collections to be ordered, if required.

The following table describes the `collects` table.

Table 342. collects table

Column name	Type	Constraints	Description
collectingEntityId	32-bit integer	Foreign key Not null	Foreign key from the `entityData` table. Specifies the collection instance. There is a row in this table for each entity within this collection.
collectedEntityId	32-bit integer	Foreign key Not null	Foreign key from the `entityData` table. Specifies an entity within the collection.
sequence	32-bit integer		For ordered collections, specifies the sequence number of the collection.

Related concepts

Collection data

Collection data defines logical collections. Collections are defined in the `collects` table. Examples of logical collections defined within NCIM include MPLS VPNs, global VLANs, and subnets.

Related reference

ncimCache.collects table

The ncimCache.collects table lists all the entities participating in a given collection.

connectActions

The connectActions table records all manual connection additions and all connection removals, including removal of connections that were discovered rather than manually added.

The following table describes the `connectActions` table.

Table 343. connectActions table

Column name	Type	Constraints	Description
connectActionsId	32-bit integer	Primary key Automatically incremented Not null	Unique auto generated primary key column.
aEndEntityId	32-bit integer	Foreign key Not null	Foreign key to the entityData table. EntityId of the A-end of the connection.
zEndEntityId	32-bit integer	Foreign key Not null	Foreign key to the entityData table. EntityId of the Z-end of the connection.

Table 343. connectActions table (continued)			
Column name	**Type**	**Constraints**	**Description**
topologyEntityId	32-bit integer	Foreign key Not null	Foreign key to the entityData table. EntityId of the topology for this connection.
unidirectional	Boolean	Not null Not null	Boolean indicating if the connection ID is directed or not.
action	6-character string	Enumeration of values “add” and “delete” Not null	Indicates an action that added or deleted an entity.
changeTime	Timestamp	Not null	Indicates when this entityAction was last updated.
username	64-character string	Not null	Indicates the user that performed the entity action.
location	512-character string		Indicates location from which user is logged in.
description	512-character string		Textual description of reason for the connect action.
userSpeed	64-bit integer		User specified speedValue from connectSpeeds table.
manual	Boolean	Not null	Indicates if the entity was manually added.

Related reference

ncimCache.connectActions table

The ncimCache.connectActions table lists all changes made manually to connections in the topology.

connects

The connects table stores data on connectivity between devices. This table belongs to the category *collections*.

Each row in the `connects` table defines a relationship between two entities.

The `connects` table stores each connection as a single record. However, because two entities are involved in a connection, the order of the connected entities within the `connects` table is random. One of the devices involved in the connection is considered to be at a notional start (or *aEnd*) of the connection, while the second device is considered to be at a notional end (or *zEnd*) of the connection.

The following table describes the `connects` table.

Table 344. connects table			
Column name	**Type**	**Constraints**	**Description**
connectionId	32-bit integer	Primary key Not null Automatically incremented Unique	This is an automatically-incremented field that must be unique for each connection across all domains.
aEndEntityId	32-bit integer	Foreign key Not null	The entityId of an interface from the `entityData` table giving the notional start of the connection.
zEndEntityId	32-bit integer	Foreign key Not null	The entityId of an interface from the `entityData` table giving the notional end of the connection.
unidirectional	Boolean	Not null Takes one of the following values: `0`: is not uni directional `1`: is uni directional	Indicates whether the connection is unidirectional or bidirectional. Currently all connections are bidirectional.

Related reference

ncimCache. connectstable

The ncimCache.connects table describes the type and speed of connections between devices.

connectSpeeds

The connectSpeeds table stores data on connectivity speed between devices.

The `connectSpeeds` table stores each connection as a single record.

The following table describes the `connectSpeeds` table.

Table 345. connectSpeeds table			
Column name	**Type**	**Constraints**	**Description**
connectionId	32-bit integer	Primary key Foreign key Not null	The connection ID from the NCIM connects table.

Table 345. connectSpeeds table (continued)			
Column name	**Type**	**Constraints**	**Description**
speedType	9-bit integer	Primary key Not null	The type of the speed of the connection. This column can take one of the following values: **DEFAULT** The standard speed derived from the data from the network element. **RATELIMIT** Not currently used. **USER** The speed specified by the user.
speedValue	64-bit integer	Foreign key	The speed of the connection, if known, in bits per second.

Related reference

ncimCache. connectstable

The ncimCache.connects table describes the type and speed of connections between devices.

contains

The `contains` table stores data on physical and logical containment. This table belongs to the category *containment*.

The following table describe the `contains` table.

Table 346. contains table			
Column name	**Type**	**Constraints**	**Description**
containingEntityId	32-bit integer	Foreign key Not null	The identifier of the containing entity from the `entityData` table
containedEntityId	32-bit integer	Foreign key Not null	The identifier of the contained entity from the `entityData` table
upwardConnection	Boolean	Unique within domain Takes one of the following values: `0`: bi directional `1`: uni directional	Used by the root-cause analysis (RCA) engine (a plug-in to the Event Gateway, ncp_g_event). This field enables the RCA engine to describe the connectivity between the entity identified by the containedEntityId field and other entities that share the same containing entity.

Related reference

ncimCache. containstable

The ncimCache.contains table lists containment information for a device.

localVlan
The localVlan table specifies which global VLAN the *local VLAN* belongs to. A local VLAN represents all the interfaces on a single chassis device that belong to a global VLAN.

dependency

The dependency table defines a general dependency between two entities. This table belongs to the category *dependency*.

This relationship is more general than the containment and connectivity relationships defined in other tables.

The following table describes the dependency table.

Table 347. dependency table

Column name	Type	Constraints	Description
independentEntityId	32-bit integer	Foreign key Not null	The identifier from the entityData table of the entity on which the dependent entity depends.
dependentEntityId	32-bit integer	Foreign key Not null	The identifier of the dependent entity from the entityData table.
dependencyType	Integer	Not null	The value identifying the type of dependency relationship.

Related reference

ncimCache.dependency table
The ncimCache.dependency table lists entities that are dependent on other devices.

deviceFunction

The deviceFunction table stores data on device vendors, associated device models together with the role of the device model such as router, switch, and so on. This table belongs to the category *entities*.

The following table describes the deviceFunction table.

Table 348. deviceFunction table

Column name	Type	Constraints	Description
vendorName	100-character string	Not null	The name of a device vendor.
vendorOID	100-character string	Not null	The MIB object ID (OID) associated with this vendor.
sysObjectId	100-character string	Primary key Not null	The vendor's authoritative identification of the network management subsystem contained in entities of this type. Provides an indication of what kind of device this is.
deviceModel	150-character string	Not null	The commercial name associated with this device type.

Table 348. deviceFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
deviceFunction	30-character string		The function of the device, such as "Router," "Switch," "Hub," or "Firewall."

discoverySource

The `discoverySource` table describes the source of the data discovered for an entity. In the case where there are multiple sources, for example, SNMP and an EMS, or two different EMSs, the table identifies all of the data sources for that entity.

In the case where there are multiple data sources for a single entity, there will be multiple rows in the `discoverySource` table for that entity. The nativeId and nativeIdString fields are important, as they are the identifiers used by the EMS to identify a given device. Network Manager might refer to the same entity in a completely different way. For example, a DNS lookup on a retrieved IP might have provided a DNS name, and this DNS name would then be used to name the entity in the NCIM topology database.

The following table describes the `discoverySource` table.

Table 349. discoverySource table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of an entity from the `entityData` table.
managedBy	255-character string		The name of an element manager. This is usually either Network Manager or the name of an EMS.
source	14-character string		Source of the data. This field takes one of the following values: • Unknown • Other • TopologyEditor • PresetLayer • Agent • Collector

Table 349. discoverySource table (continued)			
Column name	**Type**	**Constraints**	**Description**
discoveryProtocol	13-character string		Protocol of the data provided by this discovery source. This field takes one of the following values: • Unknown • Other • Manual • FlatFile • SNMP • Telnet • XML-RPC • VSphere • OtherJavaAPI • TL1 • CORBA
nativeId	Integer		Identifier used by the discovery source to identify a given device.
nativeIdString	255-character string		String used by the discovery source to identify a given device.
managedElementId	255-characters		String optionally used to hold a non-EMS specific ID for a device. Useful where multiple systems need to manage the device but might use different EMSs to do so.

domainMembers

The `domainMembers` table stores information on membership of entities within domains. This table belongs to the category *domains*.

The following table describes the `domainMembers` table.

Table 350. domainMembers table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null Automatically incremented	Foreign key to the entityNameCache table.
domainMgrId	32-bit integer	Foreign key Not null	The identifier of the domain from the domainMgr table.

Related concepts

entityData table and entity view

Information on entities is held in the entityData table in Network Manager versions 3.9 and later. This table replaces the entity table used in earlier versions. To ensure backward compatibility an entity view has been created to hold the same data as the entity table from earlier versions.

Related reference

ncimCache.domainMembers table

The ncimCache.domainMembers table shows the domain to which an entity belongs.

domainMgr

The domainMgr table stores data on network domains. This table belongs to the category *domains*.

The following table describes the domainMgr table.

Table 351. domainMgr table

Column name	Type	Constraints	Description
domainMgrId	32-bit integer	Primary key Not null Automatically incremented Unique	This is an automatically-incremented field that must be unique for each domain.
domainName	255-character string	Not null	The name of the domain.
creationTime	Timestamp	Not null	The timestamp indicating when the domain was created.
lastUpdated	Timestamp	Not null	The timestamp indicating when this domain was last updated.
managerName	100-character string	Foreign key Not null	The application that manages this domain. This is usually Network Manager. This is one of the network manager applications specified in the `manager` table.
description	255-character string		The textual description of the domain.
webtopDataSource	32-character string		The name of theTivoli Netcool/OMNIbus Web GUI data source that Topoviz must connect to. The default value is `NCOMS`. If you change this value, then you must also edit the `ModelNcimDb.`*`DOMAIN`*`.cfg` and insert the new value there.
domainHost	32-character string		Used by Topoviz to communicate with the Network Manager server. This value is automatically set by the ncp_model process.

Table 351. domainMgr table (continued)			
Column name	**Type**	**Constraints**	**Description**
domainPort	32-character string		Used by Topoviz to communicate with the Network Manager server. This value is automatically set by the ncp_model process.
batchUpdatePercent	8-bit integer		This field is updated by the domain manager during batch updates.

Related concepts

entityData table and entity view

Information on entities is held in the entityData table in Network Manager versions 3.9 and later. This table replaces the entity table used in earlier versions. To ensure backward compatibility an entity view has been created to hold the same data as the entity table from earlier versions.

entityActions

The entityActions table records all manual node additions and all node removals, including removal of nodes that were discovered rather than manually added and the swapping of nodes into and out of a domain.

The following table describes the `entityActions` table.

Table 352. entityActions table			
Column name	**Type**	**Constraints**	**Description**
entityActionsId	32-bit integer	Primary key Automatically incremented Not null	Unique auto generated primary key column.
entityId	32-bit integer	Foreign key Not null	Foreign key to the entityData table.
domainMgrId	32-bit integer	Foreign key Not null	Foreign key to the domainMgr. Indicates domain that an entity was added to or deleted from.
action	6-character string	Enumeration of values “add” and “delete" Not null	Indicates an action that added or deleted an entity.
changeTime	Timestamp	Not null	Indicates when this entityAction was last updated.
username	64-character string	Not null	Indicates the user that performed the entity action.
location	512-character string		Indicates location from which user is logged in.
description	512-character string		Textual description of reason for the entity action.

Table 352. entityActions table (continued)			
Column name	**Type**	**Constraints**	**Description**
manual	Boolean	Not null	Indicates if the entity was manually added.

Related reference
ncimCache.entityActions table
The ncimCache.entityActions table lists all devices added using the manual topology API.

entityClass

The entityClass table stores information on all device classes and relationships between device classes. The table belongs to the category *entities*.

The following table describes the entityClass table.

Table 353. entityClass table			
Column name	**Type**	**Constraints**	**Description**
classId	32-bit integer	Primary key Not null Automatically incremented Unique	A unique field for each class.
className	32-character string	Not null	The name of a class of devices.
superClassId	32-bit integer	Foreign key	The classID value associated with the class that contains the device class specified by the className value. For example, the classId for the NetworkDevice class is 5. Because Alcatel and Cisco device classes are contained in the NetworkDevice device class, they have a superClassId of 5.
classType	32-character string	Not null	The type of device or type of class.
managerName	64-character string	Foreign key Not null	The application that manages this device class. This is usually Network Manager.

Related reference
physicalChassis

The physicalChassis table stores the attributes of chassis entities.

entityData

The entityData table stores data on entities. This table belongs to the category *entities*.

The following table describes the entityData table.

Table 354. entityData table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null Automatically incremented Unique	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.
mainNodeEntityId	32-bit integer	Foreign key	This field is relevant only for entities that are wholly contained within a single main node device. The field therefore has a non-null value only for entities that are related to a single main node device, such as the main node itself, physical and logical device components, or logical interfaces (for example IP end points or local VLAN entities).
entityName	255-character string	Not null Unique within domain	Name of the entity. This field must be unique for all entities within a given domain.
entityType	32-bit integer	Foreign key Not null	A lookup value that indicates the type of entity. To look up the entity type, use the entityType table.
createTime	Timestamp	Not null	Indicates when the entity was created.
changeTime	Timestamp	Not null	Indicates when this entity was last updated.
displayLabel	255-character string	Not null	Human-readable name to be displayed adjacent to this entity in a topology map and in the **Network Views** tabular layout.
description	512-character string		Textual description of the entity.
alias	255-character string		Field that can be used to store a user-defined name for the entity.

Table 354. entityData table (continued)

Column name	Type	Constraints	Description
manual	Boolean	Not null Takes one of the following values: • 0: entity was not manually added • 1: entity was manually added Default = 0	Indicates whether this entity was discovered as part of the discovery process or was manually added.
cdmAdminState	Enumerated value 16-bit integer	Takes one of the following values: • 0 - Unknown: administrative state of this element is not known. • 1 - Other: administrative state of this element is known, but does not match one of the defined enumerated values. • 2 - Enabled: this entity has been enabled for use. • 3 - Disabled: this entity has been disabled and cannot be used. Default value = 0	An enumeration that corresponds to the AdminState attribute in the cdmModelObject view. The values of this are stored in the enumerations table under the cdmAdminState group.

Related concepts

entityData table and entity view

Information on entities is held in the entityData table in Network Manager versions 3.9 and later. This table replaces the entity table used in earlier versions. To ensure backward compatibility an entity view has been created to hold the same data as the entity table from earlier versions.

Related reference

ncimCache.entityData table

The ncimCache.entityData table holds different kinds of data about entities.

entityDetails

The entityDetails table allows the addition of arbitrary data about an entity in the form of key-value pairs. This enables you to extend the database to provide additional data on entities. The entityDetails table belongs to the category *entities*.

To add arbitrary data about an entity to the entityDetails table, you must first customize your discovery to retrieve data from an external source, using the IP address of the device as a lookup.

Restriction: NCIM cannot check the constraints of any value that is stored with the key in the entityDetails table.

On completion of a new discovery, MODEL automatically populates the NCIM topology database. You can modify the way in which this population occurs in order to ensure that the customer data held in the ExtraInfo section of the interface records within the MODEL database is used to populate key-value pair records within the entityDetails table.

The following table describes the entityDetails table.

Table 355. entityDetails table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier from the `entityData` table of an entity. The current row of this table provides key-value pair data for this entity.
keyName	255-character string	Not null	Name of the key part of the extra data. For example, this might be the name of the customer or location associated with this device.
keyValue	1000-character string		Value of the key part of the extra data. For example, this might be a customer type.

entityNameCache

The entityNameCache table is a lookup table that provides the entity name for every entity in the `entityData` table. This table belongs to the category *entities*.

The following table describes the entityNameCache table.

Table 356. entityNameCache table

Column name	Type	Constraints	Description
entityId	32-bit integer	Primary key Not null	Automatically-incremented field that provides a unique value for each entity across all domains.
entityName	255-character string	Not null	The name of the entity.
domainMgrId	32-bit integer	Not null	The identifier from the domainMgr table of the domain that contains this entity.

entityType

The entityType table provides a comprehensive list of every entity type in NCIM. It belongs to the category *entities*.

If you want to define a new entity type, you must update the entityType table to include the new entity type.

The following table describes the entityType table:

Table 357. entityType table

Column name	Type	Constraints	Description
entityType	32-bit integer	Primary key Not null	Unique field for each entity type.
typeName	32-character string	Not null	The name of an entity type.
dbTable	32-character string	Not null	The name of the NCIM database table that contains attribute data for this entity type.
metaClass	Enumerated value	Not null Takes one of the following values: • Element • Collection • Protocol EndPoint • Topology • Service • NetworkPipe • Attribute	The category of device to which this entity type belongs.
managerName	64-character string	Not null	The application that manages this entity type. This is usually PrecisionIP (Network Manager).

The following table summarizes the information in the entityType table. You can use this table to look up the value for a particular type of entity, for example, for defining a connectivity type for use in the Hop View and Network Views.

Related concepts

Entities

A Network Manager discovery returns many different types of entity. If you understand the entities that you might encounter, you can more easily restrict your queries to return only required information.

Related reference

config.serviceTypes table

The config.serviceTypes table contains configuration information for the SAE plug-in.

enumerations

The `enumerations` table provides a means of looking up a human-readable string value by providing a numerical value. Each enumeration is defined as a key-value pair. The `enumerations` table belongs to the category *mapping*.

Description

The following table describes the `enumerations` table.

Table 358. enumerations table

Column Name	Type	Constraints	Description
enumGroup	100-character string	Primary key with enumKey Not null	Groups together all the key-name pairs that form part of a single enumeration.
enumKey	32-bit integer	Primary key with enumGroup Not null	Integer value used as the key by the enumeration.
enumValue	100-character string	Not null	Human-readable string value that corresponds to the key.
enumDescription	100-character string		Brief description of the enumeration.

Summary

The following table summarizes the information in the `enumerations` table.

Table 359. Summary of the information in the enumerations table

Enumeration group	Description	Example
accessProtocol	Address protocol.	3 = Ipv6
altitudeUnits	Positional data unit.	5 = Miles
ASN	BGP ASN assignment lookups.	8406 = Cablecom
cardIfConnector TypeEnabled	Interface connector type.	3 = RJ45
cdmAdminState	CDM style administrative state of the entity.	2 = Enabled
cdmCardConfiguration	CDM style card configuration state.	2 = Configured
cdmDataLinkLayerDiscovery	CDM style data link layer protocol.	4 = LLDP
cdmDuplex	CDM style duplex state.	2 = Auto
cdmEncapsulation	CDM-style encapsulation setting.	17 = frame-relay
cdmIpAddressType	CDM-style IP address type.	7 = Multicast
cdmMediaType	CDM-style media type.	3 = gbic
cdmPhysType	CDM-style physical entity type.	8 = sensor

Table 359. Summary of the information in the enumerations table (continued)

Enumeration group	Description	Example
cdmSlotState	CDM-style slot state.	3 = connected
cdmSwitchPortMode	CDM-style switch port mode.	3 = Trunk
cefcFRUPowerAdmin Status	Administrative status for a PSU	1 = on
cefcFRUPowerOper Status	Operational status of a PSU	8 = failed
cefcModuleAdmin Status	Administrative status of a card	1 = enabled
cefcModuleOper Status	Operational status of a card	2 = ok
cefcModuleReset Reason	Reason for card reset	2 = powerUp
cefcPower RedundancyMode	Redundancy mode of PSUs	2 = redundant
cpwOperStatus	Operational status of a virtual circuit.	1 = up , Ready to pass packets
cpwVcType	Virtual circuit type.	9 = atmVccCell
discoveryProtocol	Method used for network discovery.	5 = Telnet
discoverySource	Source of network discovery data.	5 = Collector
dot3StatsDuplexStatus	Duplex type.	3 = FullDuplex
duplexToLegacyNcim	Legacy duplex method used with NCIM.	1 = fullDuplex
entPhysicalClass	Entity MIB type of an entity	10 = port
entPhysicalIsFRU	Indication of whether a component is field replaceable	1 = true
entSensorScale	Scale of a sensor	11 = mega
entSensorStatus	Status of a sensor	1 = OK
entSensorThreshold Evaluation	Indication of whether to evaluate sensor threshold-crossing	1 = true
entSensorThreshold NotificationEnable	Indication of whether threshold-crossing should be notified	1 = true
entSensor ThresholdRelation	Sensor relationship type to evaluate	1 = lessThan
entSensor ThresholdSeverity	Severity of a sensor threshold-crossing	20 = major
entSensorType	Type of sensor	3 = voltsAC
ifAdminStatus	Administrative status of an interface	1 = up
ifOperStatus	Operational status of an interface	1 = up

Table 359. Summary of the information in the enumerations table (continued)		
Enumeration group	**Description**	**Example**
ifOperStatusToOperationalStatus	Mapping used to translate SNMP ifOperStatus values into CDM OperationalStatus style values	1 = started
ifType	Type of interface	24 = softwareLoopback
ipForwarding	Is IP forwarding on?	2 = not-forwarding
jnxVpnPwStatus	Status of the JNX VPN.	2 = up
mscType	MSC type.	3 = MSCS
ncimDuplexToCdm	Duplex method used from NCIM to CDM	2 = fullDuplex
OperationalStatusEnum	Operational status	2 = started
ospfIfState	OSPF interface state.	2 = loopback
ospfIfType	OSPF interface type	3 = pointToPoint
protocolEndPoint	Protocal endpoint value	3 = layer3
ranTechnologyType	Radio access network technology type	4 = UMTS
sysServices	Open Systems Interconnection (OSI) layers supported by a device	5 = physical(1) network(3)
terminationPointType	Type of termination type.	2 = ctp , connection termination point
transceiverType	Type of transceiver.	3 = Dedicated
TruthValue	Boolean values.	2 = 0 , false
TruthValueString	Textual equivalent of Boolean values.	2 = false

hostedService

A *hosted service* is a service or application running on a specific main node device. The hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

The following table describes the hostedService table.

Table 360. hostedService table			
Column name	**Type**	**Constraints**	**Description**
hostingEntityId	32-bit integer	Foreign key Not null	The identifier of a main node device from the `entityData` table. This main node device hosts the service or application identified by hostedEntityId.

Table 360. hostedService table (continued)			
Column name	**Type**	**Constraints**	**Description**
hostedEntityId	32-bit integer	Foreign key Not null	The identifier of a service or application from the `entityData` table. This service or application is running on the main node device identified by hostingEntityId.

Related reference

ncimCache.hostedService table
The ncimCache.hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

bgpService
The bgpService table represents a BGP service and includes relevant protocol data. This BGP service runs on a device, as modeled in the hostedService table.

ospfService
The ospfService table represents an OSPF service and includes relevant protocol data. This OSPF service runs on a device, as modeled in the hostedService table.

manager

The `manager` table lists the applications that manage the network domains stored in NCIM, for example Network Manager. The `manager` table belongs to the category *domains*.

The following table describes the `manager` table.

Table 361. manager table			
Column name	**Type**	**Constraints**	**Description**
managerName	64-character string	Primary key Not null	The textual identifier of one of the network management applications that manage the network domains stored in NCIM.
description	255-character string	Not null	Full name and descriptive information of the network management application.
version	32-character string	Not null	The current version of the network management application.
contact	64-character string		Contact person for the network management application.

mappings

The `mappings` table provides a means of looking up an alternative textual name. It is used to map non-human-readable data to human-readable data. The `mappings` table belongs to the category *mapping*.

Description

The following table describes the `mappings` table.

Table 362. mappings table			
Column name	**Type**	**Constraints**	**Description**
mappingGroup	100-character string	Primary key with mappingkey Not null	Groups together all the key-value pairs that form part of a single mapping type.
mappingkey	100-character string	Primary key with mappingGroup Not null	Non-human readable string value used as the key by the enumeration.
mappingValue	100-character string	Not null	Human-readable string value that corresponds to the key.
mappingDescription	1000-character string		Description of the item specified by the key-value pair.

Summary

The following table summarizes the information held in the `mappings` table.

Table 363. Summary of information held in the mappings table		
Mapping group	**Description**	**Example**
entPhysicalVendorType	Physical component lookups	1.3.6.1.4.1.9.12.3.1.9.20 .33 = cevCat8500m4pDs3
IANAEnterprise	Internet Assigned Numbers Authority (IANA) vendor object Identifier (OID) to vendor name	1.3.6.1.4.1.9= Cisco Systems, Inc
MACVendors	Holds partial MAC addresses that represent the Organizationally Unique Identifier (OUI)	00:02:9C = 3COM
sysObjectId	Lookups for sysObjectId to device model	1.3.6.1.4.1.318.1.3.2.6 = APC SmartUPS 2000

Related reference

physicalChassis
The `physicalChassis` table stores the attributes of chassis entities.

networkPipe

The networkPipe table represents managed connections in the network. This table belongs to the category *connectivity*.

The following table describes the networkPipe table.

Table 364. networkPipe table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a network pipe entity from the `entityData` table.

Table 364. networkPipe table (continued)			
Column name	**Type**	**Constraints**	**Description**
connectionId	32-bit integer	Foreign key Not null	The identifier of a connection from the `connects` table.
aggregationType	Enumerated value	Not null Takes one of the following values: 1: unknown 2: no lower level 3: in parallel 4: in sequence	Indicates how the network pipe is made up of other network pipes. You can model redundancy by combining lower-level network pipes in parallel.

Related reference

ncimCache.networkPipe table
The ncimCache.networkPipe table represents managed connections.

entityDetails
The entityDetails table allows the addition of arbitrary data about an entity in the form of key-value pairs. This enables you to extend the database to provide additional data on entities. The entityDetails table belongs to the category *entities*.

pipeComposition
The pipeComposition table allows a higher-level connection to be defined in terms of its lower-level connections. This table belongs to the category *connectivity*.

Hierarchy modeling with the networkPipe and pipeComposition tables
The networkPipe table and pipeComposition table can be used together to represent connectivity at different layers, for example the modeling of layer 2 and layer 3 connections.

notes

The `notes` table provides a means of storing textual data related to an entity. This table belongs to the category *entities*.

The following table describes the `notes` table.

Table 365. notes table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of an entity from the `entityData` table.
createTime	Timestamp	Not null	The date and time of entity creation.
note	1000-character string		A textual note associated with the entity.

pipeComposition

The pipeComposition table allows a higher-level connection to be defined in terms of its lower-level connections. This table belongs to the category *connectivity*.

The pipeComposition table can be used together with the networkPipe table to represent a hierarchy of connections.

The following table describes the pipeComposition table.

Table 366. pipeComposition table			
Column name	**Type**	**Constraints**	**Description**
groupComponent	32-bit integer	Foreign key Primary key with partComponent Not null	A foreign key from the row in the `entityData` table that represents the network pipe instance.
partComponent	32-bit integer	Foreign key Primary key with groupComponent Not null	A foreign key from the row in the `entityData` table that represents the network pipe that is part of the composition.
aggregationSequence	32-bit integer		The sequence number of the composition. For unordered paths this value can be null.

Related reference

ncimCache.pipeComposition table
The ncimCache.pipeComposition table can be used with the networkPipe table to represent a hierarchy of connections.

networkPipe
The networkPipe table represents managed connections in the network. This table belongs to the category *connectivity*.

Hierarchy modeling with the networkPipe and pipeComposition tables
The networkPipe table and pipeComposition table can be used together to represent connectivity at different layers, for example the modeling of layer 2 and layer 3 connections.

probeTooltip

The `probeTooltip` view joins data from the `entityData`, `probeEndPoint`, and `probe` tables.

The information stored in this view is accessible from the tooltip by hovering over a link between two SLA Probe entities on a network map.

The following table describes the `probeTooltip` view.

Table 367. probeTooltip view		
Column name	**Description**	**Containing table**
entityid	The entity ID of the source or destination probe protocol end point.	entityData
role	The role of this particular endpoint in the related probe configuration. Takes one of the following values: • other • source • target	probeEndPoint
probeEntityId	The entity ID of the probe entity.	probeEndPoint

Table 367. probeTooltip view (continued)

Column name	Description	Containing table
probeid	Unique probe identifier. Identifies the probe uniquely at a device level.	probe
nativetypeid	An integer representation of the probe type. The values depend on the data definition in the probe.	probe
nativetype	A human-readable textual representation of the `nativeTypeId`.	probe
adminstatus	The administrative status of the probe. Takes one of the following values: • active • inactive • other	probe
operstatus	The operational status of the probe. Takes one of the following values: • active • inactive • other	probe
frequency	Duration between probe operations, in seconds.	probe
timeout	The maximum time to wait for a proper operation to complete, in ms.	probe
name	Name of the probe.	probe
owner	Owner/creator of the probe instance.	probe
target	Destination address. Can be an IP address.	probe
source	Source address. Can be an IP address.	probe
sourceinterface	Source interface index.	probe
vrfname	VRF associated with this probe.	probe
differentiatedservice	Type of service octet value (if set in IP header).	probe
probecount	Number of packets to transmit.	probe
targetport	Port number on the target.	probe

Table 367. probeTooltip view (continued)		
Column name	**Description**	**Containing table**
sourceport	The port number on the source.	probe
sourcevoiceport	Specifies the voice port on the gateway.	probe
packetinterval	The delay between packets, in ms.	probe
codectype	Codec type used by Jitter probes. Takes one of the following values: • unknown • g711Alaw • g711Ulaw • g729A	probe
icpifadvantage	Used in Jitter probe ICPIF calculations.	probe
httpversion	HTTP server version. Used with HTTP probes.	probe
callduration	Duration for RTP/Video probes.	probe

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

The following table describes the protocolEndPoint table.

Table 368. protocolEndPoint table			
Column name	**Type**	**Constraints**	**Description**
endPointEntityId	32-bit integer	Foreign key Not null	The identifier of an entity from the `entityData` table that specifies protocol-specific addressing information for this endpoint.
implementingEntityId	32-bit integer	Foreign key Not null	The identifier of an entity from the `entityData` table that implements this protocol end point. This is usually a device interface.

Related reference

ncimCache.protocolEndpoint table

The ncimCache.protocolEndpoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

atmEndPoint

The atmEndPoint table represents a logical ATM end point and includes relevant ATM data. This endpoint is implemented by a physical interface.

bgpEndPoint
The bgpEndPoint table represents a logical BGP end point and includes relevant BGP data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table

igmpEndPoint
The igmpEndPoint table holds information on the Internet Group Membership Protocol (IGMP) End Points.

ipMRouteEndPoint
The ipMRouteEndPoint table holds information on the IP Multicast Routing Protocol End Points.

mplsTETunnelEndPoint
The mplsTETunnelEndPoint table represents an MPLS TE protocol end point and is implemented on the interface associated with the configured tunnel. The end point references the associated TE tunnels unique instance id.

ospfEndPoint
The ospfEndPoint table represents an OSPF end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

pimEndpoint
The pimEndPoint table represents the Protocol Independent Multicast (PIM) end points discovered in the network and their associated attributes.

portEndPoint
The portEndPoint holds data about TCP/UDP endpoints found by the NMAPScan agent.

vlanTrunkEndPoint
The vlanTrunkEndPoint table represents a VLAN trunk end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

vpwsEndPoint
The vpwsEndPoint table represents a VPWS end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

frameRelayEndPoint
The frameRelayEndPoint table represents a logical Frame Relay end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

ipEndPoint
The ipEndPoint table represents an IP end point and includes relevant data. The endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

Protocol endpoint tables
The protocolEndPoint and ipEndPoint tables can be used in SQL queries to identify the IP addresses that are implemented by the device interfaces.

topologyLinks

The topologyLinks table allows you to identify which connections belong to a specific type of topology. This table belongs to the category *connectivity*.

Examples of distinct network topologies modeled in NCIM include:

- Layer 2 topology
- Layer 3 router links: This refers to connections between routers, and therefore, between subnets.
- Pseudowire topology
- OSPF topology

The following table describes the topologyLinks table.

Table 369. topologyLinks table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a topology type entity from the `entityData` table.
connectionID	32-bit integer	Foreign key Not null	The identifier of a connection from the `connects` table.
manual	Boolean	Not null Takes one of the following values: • 0: entity was not manually added • 1: entity was manually added Default = 0	Indicates whether this entity was discovered as part of the discovery process or was manually added.

Core views

The core views group together useful entity data that does not appear in a single table.

discoveryOverview

The `discoveryOverview` view joins data from the `entityData` collates timestamps from various sources.

The information stored in this view is accessible from the **Show Discovery Overview** right-click tool in the topology GUIs.

For more information on the **Show Discovery Overview** right-click tool, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

The following table describes the `discoveryOverview` view.

Table 370. discoveryOverview view		
Column name	**Description**	**Containing table**
className	Class of devices to which this device belongs.	entityClass
currentTime	The current time is given as a visual reference aid only.	NA
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData
entityName	IP address, DNS name, or sysName for this device. For example, an IP address such as 172.20.1.7, or a DNS name, such as company-abc.net.	entityData

Table 370. discoveryOverview view (continued)		
Column name	**Description**	**Containing table**
interfaceFiltered	A flag to show whether the device has had interface filtering applied to it or not. If you do not see all the information for the device that you expect, the information might have been filtered out. You can do an SNMP walk of the device using the MIB Browser with the option to ignore filtering.	discoveryAttributes
ipAddress	IP address through which this entity was discovered and through which it is monitored.	physicalChassis
firstDiscoveryComplete	Date and time when the entity was first uploaded to the NCIM topology database.	entityData
lastDiscovered	Date and time when the Details agent last accessed the device.	physicalChassis
lastModified	Date and time when the last detected change on the device was reflected in the NCIM topology database. For example, if an interface name on the device changes, this is the time at which that change was uploaded to NCIM.	entityData
rebootTime	Date and time when the device was last rebooted. This is calculated based on the sysUpTime MIB value retrieved from the device, so **Last Reboot** is only available if the sysUpTime was retrieved. For example, for devices with no SNMP access, the value of **Last Reboot** is NULL.	Calculated based on data in the chassis table

entity

The `entity` view joins data from the `entityData` and `domainMembers` tables and is equivalent to the `entity` table that existed in Network Manager versions 3.8 and earlier.The `entity` view stores data on entities and includes the domainMgrId, which the domain in which the entity is located.

The following table describes the `entity` view.

Table 371. entity view		
Column name	**Description**	**Containing table**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData
mainNodeEntityId	This field is relevant only for entities that are wholly contained within a single main node device. The field therefore has a non-null value only for entities that are related to a single main node device, such as the main node itself, physical and logical device components, or logical interfaces (for example IP end points or local VLAN entities).	entityData
entityName	Name of the entity. This field must be unique for all entities within a given domain.	entityData
domainMgrId	The identifier of the domain from the domainMgr table.	domainMgr
entityType	A lookup value that indicates the type of entity. To look up the entity type, use the entityType table.	entityData
createTime	Indicates when the entity was created.	entityData
changeTime	Indicates when this entity was last updated.	entityData
displayLabel	Human-readable name to be displayed adjacent to this entity in a topology map and in the **Network Views** tabular layout.	entityData
description	Textual description of the entity.	entityData
alias	Field that can be used to store a user-defined name for the entity.	entityData
manual	Indicates if the entity was manually added.	entityData

Related concepts

entityData table and entity view

Information on entities is held in the entityData table in Network Manager versions 3.9 and later. This table replaces the entity table used in earlier versions. To ensure backward compatibility an entity view has been created to hold the same data as the entity table from earlier versions.

Related reference

entityType

The entityType table provides a comprehensive list of every entity type in NCIM. It belongs to the category *entities*.

interfaceDomain

The interfaceDomain view adds the domain name to the interface table. This view is used to get the domain details of a particular interface.

The following table describes the interfaceDomain view.

Table 372. interfaceDomain view

Column name	Description	Containing table
ipAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	chassis In the chassis table, this field is called accessIpAddress.
ifName	The name assigned to the interface.	interface
ifDescr	A description of the interface.	interface
entityName	Name of the entity. This field must be unique for all entities within a given domain.	entityData
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData
domainMgrId	The identifier of the domain from the domainMgr table.	domainMgr
domainName	The name of the domain in which the node was discovered.	domainMgr

interfaces

The interfaces view provides a consolidated set of data on all device interfaces.

The following table describes the `interfaces` view. The type, constraints, and description are automatically inherited, where appropriate, from the tables to which the fields belong.

Table 373. interfaces view

Column name	Description	Containing table
displayLabel	Human-readable name to be displayed adjacent to this entity in a topology map and in the **Network Views** tabular layout.	entityData
duplex	Actual duplex of the interface. Takes one of the following values: • HalfDuplex • FullDuplex • Auto • Unknown • Other	interface
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	chassis
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData
entityName	Name of the entity. This field must be unique for all entities within a given domain.	entityData
ifAdminStatus	The required state of the interface. Takes one of the following values: • Up • Down • Testing	interface
ifAlias	The alias for the interface.	interface
ifDescr	A description of the interface.	interface
ifHighSpeed	An estimate of the current bandwidth of the interface in units of 1,000,000 bits per second.	interface
ifIndex	The index of the interface.	interface
ifMTU	The maximum transmission unit for this interface.	interface

Table 373. interfaces view (continued)

Column name	Description	Containing table
ifName	The name assigned to the interface.	interface
ifOperStatus	The current operational state of the interface. Takes one of the following values: • Up • Down • Testing • unknown • dormant • notPresent • lowerLayerDown	interface
ifPhysAddress	The physical address of the interface.	interface
ifPromiscuousMode	Indicates whether this interface only accepts packets or frames addressed to this station. Takes one of the following values: • True • False	interface
ifSpeed	An estimate of the current bandwidth of the interface in bits per second.	interface
ifType	The interface type.	interface
ifTypeString	The textual string for the interface type.	interface
ipAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	chassis In the chassis table, this field is called accessIpAddress.
isTrunkPort	Indicates whether this physical interface is a VLAN trunk port.	interface

Table 373. interfaces view (continued)		
Column name	**Description**	**Containing table**
mainNodeEntityId	This field is relevant only for entities that are wholly contained within a single main node device. The field therefore has a non-null value only for entities that are related to a single main node device, such as the main node itself, physical and logical device components, or logical interfaces (for example IP end points or local VLAN entities).	entityData
portNumber	The port number for this interface on the chassis device. The method of determining the port number is dependent on the make and model of the device that is discovered. For this reason, use this value with caution.	interface
domainMgrId	The ID for the domain to which this interface belongs.	domainMgr

mainNodeDetails

The mainNodeDetails view provides a consolidated set of data on all network devices.

The following table describes the `mainNodeDetails` view. The type, constraints, and description are automatically inherited, where appropriate, from the tables to which the fields belong.

Table 374. mainNodeDetails view		
Column name	**Description**	**Containing table**
altitude	Vertical height above World Geodetic System WGS84 datum surface (EGM96) at the particular geographical location.	geographicLocation
altitudeUnits	Units to use for the altitude. • 0 – Meters • 1 – Kilometers • 2 – Centimeters • 3 - Feet • 4 – Yards • 5 - Miles • 6 - Inches	geographicLocation
className	The name of a class of devices. The master className field is in the entityClass table.	chassis

Table 374. mainNodeDetails view (continued)

Column name	Description	Containing table
classType	The type of device or type of class.	entityClass
description	Textual description of the entity.	entityData
displayLabel	Human-readable name to be displayed adjacent to this entity in a topology map and in the **Network Views** tabular layout.	entityData
domainMgrId	The ID for the domain to which this device belongs.	domainMgr
domainName	The name of the domain to which this device belongs.	domainMgr
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	chassis
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	chassis
entityId	Unique ID for each entity across all domains.	entityData
entityName	Name of the entity. This field must be unique for all entities within a given domain.	entityData
ipAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	chassis In the chassis table, this field is called accessIpAddress.

Table 374. mainNodeDetails view (continued)

Column name	Description	Containing table
ipForwarding	Indication of whether this entity is acting as an IP gateway in respect to the forwarding of datagrams received by this entity but not addressed to this entity. IP gateways forward datagrams, whereas IP hosts do not, unless the source is routed through the host. Takes one of the following values: • forwarding • not-forwarding	chassis
latitude	Measurement of latitude for this entity. This is the angular distance (north and south) from the equator on the earth's surface; for example, 78.838753. The value of this attribute is determined based on World Geodetic System WGS84 standard.	geographicLocation
locationDescription	Free-format textual description of location	geographicLocation
locationId	The location identifier by which the location is know to the end-user.	geographicLocation
longitude	Measurement of longitude for this entity. This is the angular distance (east and west) from the prime meridian on the earth's surface; for example, 35.832636. The value of this attribute is determined based on World Geodetic System WGS84 standard.	geographicLocation
manual	Indicates whether this entity was discovered as part of the discovery process or was manually added.	entityData
serialNumber	The serial number of the entity.	chassis
sysContact	The textual identification of the contact person for this managed node, and information on how to contact this person. If no contact information is known, the value is the zero-length string.	chassis

Table 374. mainNodeDetails view (continued)

Column name	Description	Containing table
sysDescr	A textual description of the entity. This value must include the full name and version identification of the system hardware type, software operating-system, and networking software.	chassis
sysLocation	The physical location of this node, for example "telephone closet, 3rd floor." If the location is unknown, the value is the zero-length string.	chassis
sysName	An administratively-assigned name for this managed node. By convention, this is the fully-qualified domain name of the node. If the name is unknown, the value is the zero-length string.	chassis
sysObjectId	The vendor's authoritative identification of the network management subsystem contained in the entity.	chassis

Table 374. mainNodeDetails view (continued)		
Column name	**Description**	**Containing table**
sysServices	A value that indicates the set of services that this entity potentially offers. The value is a sum that initially takes the value zero. Then, for each layer, L, in the range 1 through 7, that this node performs transactions for, 2 raised to (L - 1) is added to the sum. For example, a node that performs only routing functions would have a value of 4 (2^(3-1)). A node that is a host offering application services would have a value of 72 (2^(4-1) + 2^(7-1)). For the Internet suite of protocols, values should be calculated accordingly: • Layer 1: Physical, for example repeaters) • Layer 2: Datalink or subnetwork, for example bridges • Layer 3: Internet, for example supports IP • Layer 4: End-to-end, for example supports TCP • Layer 7: Applications, for example supports the SMTP For systems including OSI protocols, layers 5 and 6 can also be considered.	chassis
sysUpTime	The time (in hundredths of a second) since the network management portion of the system was last reinitialized.	chassis
timeZoneOffset	Offset of local time from UTC in format UTC-HH:MM or UTC +HH:MM; for example, UTC+10:30, UTC-6:00.	geographicLocation

interfaceDomain

The interfaceDomain view adds the domain name to the interface table. This view is used to get the domain details of a particular interface.

The following table describes the interfaceDomain view.

Table 375. interfaceDomain view		
Column name	**Description**	**Containing table**
ipAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	chassis In the chassis table, this field is called accessIpAddress.
ifName	The name assigned to the interface.	interface
ifDescr	A description of the interface.	interface
entityName	Name of the entity. This field must be unique for all entities within a given domain.	entityData
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData
domainMgrId	The identifier of the domain from the domainMgr table.	domainMgr
domainName	The name of the domain in which the node was discovered.	domainMgr

Entity attribute tables

Entity attribute tables define attributes for the entities discovered by Network Manager, that is, layer 1, layer 2 and layer 3 entities.

If the entity is a physical element, such as a chassis or module, these are the tables that contain the attributes which relate specifically to that physical element, such as sysDescr and sysObjectId for a chassis or serialNumber for a module. If the entity is a device collection, such as a VLAN or VPN, these tables contain collection-specific parameters, such as vlanId or vpnName.

aggregationGroup

The aggregationGroup table holds information about Link Aggregation Groups (LAGs).

The following table describes the aggregationGroup table.

Table 376. aggregationGroup table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Not null Primary Key Foreign Key	Automatically incremented ID that provides a unique value for each aggregated group across all domains.
lagId	32-bit integer	Not null	The ID of the LAG.

Table 376. aggregationGroup table (continued)			
Column name	**Type**	**Constraints**	**Description**
lagName	255-character string		The name of the LAG.
lagMode	8-character string		The mode that the LAG is operating in. Takes one of the following values: • lacp • static • unknown • other • disabled
lacpMode	7-character string		The Link Aggregation Control Protocol mode. Can be `active` or `passive`.
actorSystemPriority	Integer		The priority associated with the Actor's System ID.
actorSystemId	255-character string		The MAC address value used as a unique identifier for the server that contains this LAG.
actorAdminKey	Integer		Administrative value of the key for the LAG.
actorOperKey	Integer		Operational value of the key for the LAG.
partnerSystemPriority	Integer		The priority associated with the Partner's System ID.
partnerSystemId	255-character string		The MAC address value used as a unique identifier for the current protocol partner of this LAG.
partnerAdminKey	Integer		Administrative value of the key for the protocol partner.
partnerOperKey	Integer		Operational value of the key for the protocol partner.

antennaFunction

The `antennaFunction` table models the physical antenna which support eUTRAN cells and sectors.

The following table describes the antennaFunction table.

Table 377. antennaFunction table			
Column name	**Type**	**Constraints**	**Description**
entityID	Integer	FOREIGN KEY NOT NULL	The identifier of an antenna entity from the `entityData` table.

Table 377. antennaFunction table (continued)

Column name	Type	Constraints	Description
antennaSerial Number	64-character string	NOT NULL	Unique antenna identifier.
emsDistinguished Name	255-character string		Distinguished name by which the antenna is known to its element management system (EMS).
antennaHeight	Float		Height of the antenna above sea level in metres.
antennaDownTilt	Float		Antenna vertical tilt in degrees.
antennaBearing	Float		Bearing in degrees that the antenna is pointing in.
antennaMax Azimuth	Float		Maximum amount of change of azimuth the system can support. This is the change in degrees clockwise from bearing.
antennaMin Azimuth	Float		Minimum amount of change of azimuth the system can support. This is the change in degrees clockwise from bearing.
antennaHorizontal Beamwidth	Float		Power beamwidth of the antenna pattern in the horizontal plane. A value of 360 indicates an omni-directional antenna. A single integral value corresponding to an angle in degrees between 0 and 360.
antennaVerticall Beamwidth	Float		Power beamwidth of the antenna pattern in the vertical plane. A value of 360 indicates an omni-directional antenna. A single integral value corresponding to an angle in degrees between 0 and 360.
antennaLatitude	Float		Latitude of Antenna equipment associated with the sector. This is the angular distance (east and west) from the prime meridian on the earth's surface; for example, 35.832636. The value of this attribute is determined based on World Geodetic System WGS84 standard.
antennaLongitude	Float		Longitude of Antenna equipment associated with the sector. This is the angular distance (north and south) from the equator on the earth's surface; for example, 78.838753. The value of this attribute is determined based on World Geodetic System WGS84 standard.
antenna Manufacturer	64-character string		Vendor or manufacturer of the antenna.
antennaModel	64-character string		Vendor-specific antenna type.

atmEndPoint

The atmEndPoint table represents a logical ATM end point and includes relevant ATM data. This endpoint is implemented by a physical interface.

The following table describes the atmEndPoint table.

Table 378. atmEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a logical ATM end point from the `entityData` table.
VPI	32-bit integer		Virtual Path Indicator (VPI) in the ATM cell header. The VPI is used together with the Virtual Channel Identifier (VCI) to route the ATM cell.
VCI	32-bit integer		VCI in the ATM cell header. The VCI is used together with the VPI to route the ATM cell.

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

bgpAutonomousSystem

The bgpAutonomousSystem table stores data about a BGP autonomous system (AS), including number, name, and whether the AS is private.

The following table describes the bgpAutonomousSystem table.

Table 379. bgpAutonomousSystem table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP AS from the `entityData` table.
ASN	32-bit integer	Not null	The number of this BGP AS. This can be a value between 1 and 65535; the range from 64512 to 65535 is reserved for private use. Every AS has a unique AS number, which is assigned to it by an Internet Registry or a service provider.
ASName	64-character string		The name of this BGP AS.
isSingleHomed	8-bit integer		Indicates whether this AS is single-homed. A single-homed AS reaches networks outside of its domain through a single exit point.

Table 379. bgpAutonomousSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
isTransit	8-bit integer		Indicates whether this AS is a transit AS. A transit AS advertises routes that it learns from other ASs. A non-transit AS will only advertise its own routes.
isPrivate	8-bit integer		Indicates whether this AS is private.

bgpCluster

The bgpCluster table represents use of route reflectors within a BGP AS. This table is currently not used.

The following table describes the bgpCluster table.

Table 380. bgpCluster table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP AS from the `entityData` table.
clusterID	15-character string	Not null	Identifier for the BGP route reflector in the cluster when there is more than one route reflector in the local AS.

bgpEndPoint

The bgpEndPoint table represents a logical BGP end point and includes relevant BGP data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table

The following table describes the bgpEndPoint table:

Table 381. bgpEndPoint Table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP AS from the `entityData` table.
isEBGP	8-bit integer		Indicates whether this is an instance of the external version of BGP (eBGP).
isEBGPMultiHop	8-bit integer		Normally, two routers running eBGP must be physically connected. This field indicates whether there is a logical connection between two routers that are running eBGP. For example, there might be an intermediate router or interface between them.

Table 381. bgpEndPoint Table (continued)

Column name	Type	Constraints	Description
localIdentifier	15-character string		The unique identifier of the BGP peer router. This is often the router identifier, for example, an IP address. Corresponds to the bgpIdentifier MIB variable in BGP4-MIB.
peerIdentifier	15-character string		The unique identifier of the peer BGP router. This is often the router identifier, for example, an IP address. Corresponds to the bgpIdentifier MIB variable in BGP4-MIB.
peerState	20-character string		The BGP state of the connection, as stored in the bgpPeerState MIB variable in BGP4-MIB.
adminStatus	5-character string	Not null	The required state of the BGP connection.
localAddress	15-character string		The local IP address of the BGP connection for this router. Corresponds to the bgpPeerLocalAddr MIB variable in BGP4-MIB.
localPort	32-bit integer		The local port number for the TCP connection of the BGP connection of the router. Corresponds to the bgpPeerLocalPort MIB variable in BGP4-MIB.
remoteAddress	15-character string		The remote IP address of the BGP connection for this router. Corresponds to the bgpPeerRemoteAddress MIB variable in BGP4-MIB.
remotePort	32-bit integer		Remote port number for the TCP connection the BGP connection of this router. Corresponds to the bgpPeerRemotePort MIB variable in BGP4-MIB.

Table 381. bgpEndPoint Table (continued)

Column name	Type	Constraints	Description
remoteAS	32-bit integer		Remote AS number of the BGP peer connected to this router. The AS number can be a value between 1 and 65535; the range 64512 to 65535 is reserved for private use. Every AS has a unique AS number, which is assigned to it by an Internet Registry or a service provider. Corresponds to the bgpPeerRemoteAS MIB variable in BGP4-MIB.
connectRetryInterval	32-bit integer		Time interval, in seconds, for the ConnectRetry timer. Corresponds to the bgpConnectRetryInterval MIB variable in BGP4-MIB.
holdTime	32-bit integer		Maximum amount of time elapsed in seconds between receipt of successive KEEPALIVE or UPDATE messages.
holdTimeConfigured	32-bit integer		Time interval in seconds for the hold time configured for this BGP speaker with a peer. Corresponds to the bgpHoldTimeConfigured MIB variable in BGP4-MIB.
keepAlive	32-bit integer		Time in seconds for the KEEPALIVE timer established with the BGP peer.
keepAliveConfigured	32-bit integer		Time interval in seconds for the KeepAlive timer configured for this BGP speaker with a peer. Corresponds to the bgpPeerKeepAlive Configured MIB variable in BGP4-MIB.
minASOrigInterval	32-bit integer		Time interval in seconds for the MinASOriginationInterval timer. Corresponds to the bgpPeerMinASOrigination Interval MIB variable in BGP4-MIB.

Table 381. bgpEndPoint Table (continued)			
Column name	**Type**	**Constraints**	**Description**
minASRouteAdv Interval	32-bit integer		Time interval in seconds for the MinRouteAdvertisement Interval timer. Corresponds to the bgpPeerMinRoute AdvertiseMentInterval MIB variable in BGP4-MIB.

Related reference

protocolEndPoint
The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

frameRelayEndPoint
The frameRelayEndPoint table represents a logical Frame Relay end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

ipEndPoint
The ipEndPoint table represents an IP end point and includes relevant data. The endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

bgpNetwork

The bgpNetwork table holds a collection of BGP ASs.

The following table describes the bgpNetwork table:

Table 382. bgpNetwork table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP network from the `entityData` table.

bgpRouteAttribute

The bgpRouteAttribute table stores data about a given BGP route such as its next hop and prefix. These attributes affect routing decisions for the AS.

The following table describes the bgpRouteAttribute table.

Table 383. bgpRouteAttribute table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP AS from the `entityData` table.
AFI	30-character string		Address Family Identifier (AFI) information for this BGP route.
SAFI	30-character string		Subsequent Address Family Identifier (SAFI) information for this BGP route.

Table 383. bgpRouteAttribute table (continued)

Column name	Type	Constraints	Description
prefix	15-character string		IP prefix for the advertized network. Corresponds to the bgp4PathAttripAddrPrefix MIB variable in BGP4-MIB.
prefixLength	8-bit integer		CIDR prefix length of the advertized network. Corresponds to the bgp4PathAttripAddrPrefix-Length MIB variable in BGP4-MIB.
peerAddress	15-character string		IP address of the BGP peer from which this path was learned.
nextHop	15-character string		Next hop IP address of the eBGP to reach a given network. Corresponds to the bgp4PathAttrNextHop MIB variable in BGP4-MIB.
origin	15-character string		Origin of this BGP route.
localPreference	32-bit integer		Local preference of a route with respect to other routes to the same destination. Higher value indicates a preferred route. Corresponds to the bgp4PathAttrLocalPref MIB variable in BGP4-MIB.
aggregatingAddr	15-character string		Aggregating address for this route.
aggregatingAS	32-bit integer		AS number of the last BGP4 speaker that performed route aggregation. The AS number is a value between 1 and 65535; the range 64512 to 65535 is reserved for private use. Every AS has a unique AS number, which is assigned to it by an Internet Registry or a service provider. Corresponds to the bgp4PathAttrAggregatorAS MIB variable in BGP4-MIB.
ASPathSegment	100-character string		Sequence of AS path segments to a given network. Corresponds to the bgp4PathAttrASPath MIB variable in BGP4-MIB.

Table 383. bgpRouteAttribute table (continued)			
Column name	**Type**	**Constraints**	**Description**
atomicAggregate	40-character string		Indicates whether a less specific route is selected instead of a more specific route. Corresponds to the bgp4PathAttrAtomic-Aggregrate MIB variable in BGP4-MIB.
MED	Integer with up to 10 decimal digits.		Multi-Exit Discriminator (MED) value that indicates a preferred entry point into the AS for a given network. Lower MED values are preferred. Corresponds to the bgp4PathAttrMultiExitDisc MIB variable in BGP4-MIB.
bestRoute	10-character string		Indicates whether this route was chosen as the best BGP4 route.
peerType	10-character string		Peer type for this BGP route attribute.

bgpService

The bgpService table represents a BGP service and includes relevant protocol data. This BGP service runs on a device, as modeled in the hostedService table.

Each row in this table corresponds to a single hosted BGP service. BGP routers can only host one BGP service.

The following table describes the bgpService table.

Table 384. bgpService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a BGP service from the `entityData` table.
BGPVersion	6-character string	Not null	Negotiated BGP version number. Each peer negotiates this version number from the vector contained in the bgpVersion MIB variable within the BGP4-MIB.
BGPLocalAS	32-bit integer	Not null	Local AS for this BGP service.
BGPIdentifier	15-character string	Not null	Identifier for this BGP service.
RouteReflectorMode	25-character string	Not null	Router reflector mode for this BGP service.

Related reference

hostedService

A *hosted service* is a service or application running on a specific main node device. The hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

computerSystem

The `computerSystem` table represents the logical or virtual perspective of the physical device

The following table describes the `computerSystem` table.

Table 385. computerSystem table

Column name	Type	Constraints	Description
entityId	Integer	not null	The identifier of a computerSystem entity from the `entityData` table.
architecture	255-character string		Architecture of the device. Valid values are • alpha • i686 • sun4 • PowerPC® • PowerPC_POWER3 • PowerPC_POWER4 • PowerPC_POWER5 • PowerPC_604 • HP PA-RISC
assetID	255-character string		Asset identifier for this entity.
assetTag	255-character string		Asset tag for this entity.
autoStart	Tiny integer		Indicates whether the computer system can be automatically started. • TRUE: computer system can be automatically started. • FALSE: computer system cannot be automatically started.
availableMem ForAllVMs	64-bit integer		Amount of memory (in bytes) currently available for all virtual machines.

Table 385. computerSystem table (continued)

Column name	Type	Constraints	Description
bootOrder	255-character string		Boot sequence settings of a system. The colon (:) character is used to delimit multiple device names. Possible values for this attribute are as follows: • Hard Drive x (where x is optional and is an integer value starting from 0). • DVD x. • CD x. • LAN PXE (including all variants such as MBA, IGA and others).
ciCategory	255-character string		Configuration item category for this entity.
ciRole	Tiny integer		Identifies the environment, or role, in which a configuration item (CI) resides. For example, if a CI is set aside for test purposes, then this column can be set to a value of `Test`. If a role is needed that is not defined in the enumeration for ciRole, then use the value `Other`.
cpuLimit	64-bit integer		Maximum amount of CPU that can be used by this virtual machine even if there is more CPU available in the resource pool.
cpuReservation	64-bit integer		Amount of CPU that is reserved for this virtual machine.

Table 385. computerSystem table (continued)

Column name	Type	Constraints	Description
cpuSharesLevel	Tiny integer		Level of shares specified for this virtual machine. Shares define the relative priority or importance of a virtual machine within a specific Hypervisor. Each virtual machine is entitled to resources in proportion to its specified shares, bounded by its reservation and limit. A virtual machine with twice as many shares as another is entitled to twice as many resources. The values are comparable only across VMs that are virtualizing the same ComputerSystem.
cpuSharesValue	64-bit integer		Actual number of CPU shares allocated to this virtual machine or resource pool. It is used only when the level of CPU shares (defined by the CPUSharesLevel attribute) is set to the value `Custom`.
cdpDeviceId	255-character string		Device ID defining the node for the Cisco Discovery Protocol. This attribute is Cisco-specific and will move to a class representing the Cisco Computer System.
configLastUpdate	255-character string		UTC date and time when the information was last altered in the source application.
currentMemForAllVMs	64-bit integer		Amount of memory currently allocated for all virtual machines.
generalCIRole	255-character string		Environment, or role, in which a configuration item (CI) resides.

Table 385. computerSystem table (continued)

Column name	Type	Constraints	Description
isVMIDanLPAR	Tiny integer		Specifies if the computer system is a Logical PARtition (LPAR) or if the computer system is using other virtual machine types, such as a central processor complex (CPC) . An LPAR represents all IBM® mainframe and non-mainframe virtualization technologies, including zSeries, pSeries, iSeries, and DS8000®. Possible value are: • TRUE: computer system is an LPAR. • FALSE: computer system is using a virtual machine type other than an LPAR.
lastAuditState	Tiny integer		Last audit state for this device. Possible values are: • 0 Unknown • 1 Other • 2 Good • 3 No Physical CI • 4 No CMDB Record • 5 Inaccurate CMDB Record
lastAuditTime	Timestamp		Last audit time this entity.
lastLifecycleStateTime	Timestamp		Last lifecycle state time for this entity.

Table 385. computerSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
lifecycleState	Tiny integer		Lifecycle state for this device. Possible values are: • 0 Unknown • 1 Other • 2 Ordered • 3 Received • 4 In Test • 5 Tested • 6 Installed • 7 Enabled • 8 Disabled • 9 In Maintenance • 10 Retired • 11 Archived • 12 Accepted • 13 Draft • 14 Build • 15 Validate • 16 ProductionReady • 17 Production • 18 Sunset • 19 PostProduction • 20 Inventory • 21 Development • 22 Offline
manufacturer	255-character string		Vendor-specific hardware type for this entity.
memoryLimit	64-bit integer		Last audit time this device.
memoryReservation	64-bit integer		Amount of memory that is reserved for this machine.

Table 385. computerSystem table (continued)

Column name	Type	Constraints	Description
memorySharesLevel	Tiny integer		level of memory shares specified for this virtual machine. Shares define the relative priority or importance of a virtual machine. Each virtual machine is entitled to resources in proportion to its specified shares, bounded by its reservation and limit. A virtual machine with twice as many shares as another is entitled to twice as many resources.
memorySize	64-bit integer		Size of physical memory that is present in the computer system.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
namingRuleAlgorithms	255-character string		Proprietary attribute, used internally by IBM Tivoli Application Dependency Discovery Manager.
primaryMACAddress	255-character string		MAC Address of the network adapter in the computer system. If there are multiple network adapters in the computer system, the primary MAC Address is determined by ordering the MAC address according to their numeric value, selecting the lowest valued MAC address.

Table 385. computerSystem table (continued)

Column name	Type	Constraints	Description
processCapacityUnits	Tiny integer		Units in which the `ProcessingCapacity` field value is expressed.
processingCapacity	Floating-point number		Processing capacity of this computer system, including all of the CPUs assigned to it.
romVersion	255-character string		ROM (Read-Only Memory) or the BIOS (Basic Input/Output System) version of the motherboard in the computer system.
serialNumber	255-character string		The serial number of the entity.
serviceConsoleMemSize	64-bit integer		Amount of memory that is reserved for the service console.
signature	255-character string		Signature of the computer system. The value is composed as follows: the dot-notated IP address, the (character, the MAC Address in hexadecimal with upper case characters only (sans hashes), the) character.
swapMemSize	64-bit integer		Memory size of a datastore on which the virtual machine swap files are allocated.
systemBoardUUID	255-character string		Specifies the burned-in Globally Unique Identifier (GUID) of this piece of equipment.
systemId	255-character string		Field used by IBM Tivoli Application Dependency Discovery Manager.
cdmType	255-character string		Common data model type for this device.
uuid	255-character string		Unique identifier of this piece of equipment.

Table 385. computerSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
vmid	255-character string		Unique identifier for the virtual machine. It corresponds to the Virtual Machine ID in the VM table. For System p or System z® computer systems, this is the LPARID value.
virtual	Tiny integer		Specifies whether this computer system is virtual. Possible values are: • TRUE: computer system is virtual. • FALSE: computer system is not virtual.
vmotionEnabled	Tiny integer		Specifies whether the VMWare VMotion feature is enabled on this computer system.

controlPlaneViewCollection

The `controlPlaneViewCollection` table supports the dynamic collection views under **LTE Network Topology** > **Control Plane by Tracking Area** in the Network Views. Each instance of this entity type collects the eNodeBs in the corresponding tracking area, together with the devices that these eNodeBs are connected to on the control plane..

The following table describes the controlPlaneViewCollection table.

Table 386. controlPlaneViewCollection table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an controlPlaneViewCollection entity from the `entityData` table.
viewType	64-character string	NOT NULL	Specifies the type of view.

cpu

The cpu table describes Central Processing Units (CPUs).

The following table describes the cpu table.

Table 387. cpu table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a CPU from the `entityData` table.
serialNumber	Varchar (100)		The serial number of the processor.

Table 387. cpu table (continued)			
Column name	**Type**	**Constraints**	**Description**
modelName	Varchar (100)		The model of the processor.
manufacturer	Varchar (100)		The manufacturer of the processor.
cpuSpeed	Big int		The clock speed of the processor.
coresEnabled	Integer		How many cores are enabled.
coresInstalled	Integer		How many cores are installed.

discoveryAttributes

The discoveryAttributes table stores data that shows whether a device has been subject to interface filtering.

The following table describes the discoveryAttributes table.

Table 388. discoveryAttributes table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	Primary key Not null	The entity ID of the device.
interfaceFiltered	Boolean integer	Not null	A flag showing whether the SNMP information from the device had an interface filter applied to it. If the value is 1, then the information was filtered. The default is zero.

domainSummary

The domainSummary table stores statistical data on a given domain.

The following table describes the domainSummary table.

Table 389. domainSummary table			
Column name	**Type**	**Constraints**	**Description**
domainMgrId	32-bit integer	Foreign key Not null	An automatically-incremented field that must be unique for each domain.
createTime	Timestamp	Not null	The date and time that the domain was created.
changeTime	Timestamp	Not null	The date and time that the domain was changed.
entityCount	32-bit integer	Not null	The number of entities in the domain.
chassisCount	32-bit integer	Not null	The number of main node chassis devices in the domain.

Table 389. domainSummary table (continued)			
Column name	**Type**	**Constraints**	**Description**
interfaceCount	32-bit integer	Not null	The number of interfaces in the domain.

eirFunction

The `eirFunction` table models the Equipment Identity Register (EIR). The EIR keeps track of mobile devices which should either be banned from using the network or monitored. When a mobile phone is stolen its identity it is added to the EIR blacklist and the result is that this phone will never be able to attach to the network for service. Usually each network has its own EIR which is often combined with the HSS node. It is possible for multiple operators to share a common EIR which enables the blacklisted information to be more easily and widely available.

The following table describes the eirFunction table.

Table 390. eirFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an eirFunction entity from the `entityData` table.
eirFunctionName	64-character string	NOT NULL	Name of the eirFunction (Equipment Identity Register) instance configured on the physical node that implements the eirFunction.
MCC	3-character string		An EIR can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the EIR. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the EIR. The MCC consists of three digits.
MNC	3-character string		An EIR can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the EIR. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the EIR. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		Count of the number of PLMNs (Public Land Mobile Networks) supported by the EIR.

Table 390. eirFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
emsDistinguishedName	255-character string		Distinguished name by which the EIR is known to its element management system (EMS)
emsIpAddress	39-character string		IP address of the element management system.
vendorName	64-character string		Vendor or manufacturer of the EIR.
vendorModuleType	64-character string		Vendor specific EIR Type.
softwareVersion	64-character string		Vendor specific EIR sofware version.
operationalState	Enumeration		Operational state of the eirFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the eirFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

emsSystem

The `emsSystem` table describes EMS system entities.

The following table describes the `emsSystem` table.

Table 391. emsSystem table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of an EMS system entity from the `entityData` table.

Table 391. emsSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
collectorPort	Integer		Port number on the EMS collector used to retrieve data.
collectorhost	64-character string		EMS host identifier string
collectorSourceId	Integer		Source identifier for the collector.
emsHost	255-character string		Hostname of the EMS.
emsName	255-character string		Name of the EMS.
emsVersion	255-character string		Version of the EMS.
emsIdentifier	255-character string		Identifier for the EMS and key for integrating the Network Manager collector with the Netcool Configuration Manager driver.
emsRole	7-character string		Role of the EMS. This parameter can take one of the following values: • unknown • primary • backup • other
emsStatus	7-character string		Status of the EMS. This parameter can take one of the following values • unknown • up • down • other

enbFunction

The `enbFunction` table models the role of the eNodeB entity within a network hardware node. Multiple enbFunction instances may be implemented within a single network hardware node. The eNodeB entity manages radio air interface communication with users of the LTE network. Each eNodeB controls one or more cells which are geographic areas of radio coverage.

The following table describes the enbFunction table.

Table 392. enbFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an enbFunction entity from the `entityData` table.

Table 392. enbFunction table (continued)

Column name	Type	Constraints	Description
eNodeBId	64-character string	NOT NULL	Identifier of the eNodeB.
eNodeBName	64-character string	NOT NULL	User friendly name of the eNodeB.
MCC	3-character string		An eNodeB can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the eNodeB. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the eNodeB. The MCC consists of three digits.
MNC	3-character string		An eNodeB can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the eNodeB. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the eNodeB. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		Count of the number of PLMNs (Public Land Mobile Networks) supported by the enbFunction.
emsDistinguishedName	255-character string		Distinguished name by which the enbFunction is known to its element management system (EMS).
emsIpAddress	39-character string		IP address of the element management system.
vendorName	64-character string		Vendor or manufacturer of the enbFunction.
vendorModuleType	64-character string		Vendor-specific eNodeB type.
softwareVersion	64-character string		Vendor-specific eNodeB software version.
userCapacity	Integer		Maximum number of active pieces of user equipment (UEs) that can connect to this eNodeB simultaneously.
maximumOutputPower	Float		Maximum output power of the eNodeB.

Table 392. enbFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
operationalState	Enumeration		Operational state of the enbFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the enbFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown
backHaulConnection	39-character string		IP address of the first hop backhaul device to which the enbFunction is connected; for example, the IP address of a cell-site router.

eUtranCell

The `eUtranCell` table models a geographical area of radio coverage that is implemented and supported by physical radio equipment, such as towers, amplifiers, and antennas.

The following tables describes the eUtranCell table.

Table 393. eUtranCell table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an eUtranCell entity from the `entityData` table.
eUtranCellId	64-character string	NOT NULL	Uniquely identifies a cell within a PLMN. It is often constructed from eNodeB ID + Physical Cell ID.
eUtranCellName	64-character string	NOT NULL	Cell identifier or name of the cell.

Table 393. eUtranCell table (continued)

Column name	Type	Constraints	Description
physicalCellID	Integer		Physical cell identifier. Takes a value in the range 0 to 503. The physical cell id is used by the cell to encode and decode the data that it transmits. It is used in a similar way to the UMTS scrambling code. To avoid interference, neighboring cells should have different physical cell identifiers. The physical cell id is derived from the primary and secondary synchronization signals (PSS and SSS). The PSS takes a value from 0 to 2, the SSS takes a value from 0 to 167, and the physical cell id is determined based on the following formula: `PSS + 3*SSS` The result of this calculation equates to a value of between 0 and 503.
localCellId	Integer		Local cell id unique within the eNodeB.
emsDistinguishedName	255-character string		Distinguished name by which the eUtranCell is known to its element management system (EMS).
channelBandwidthUI	Integer		Uplink channel bandwidth. Takes one of the following values in MHz • 3 • 5 • 10 • 15 • 20
channelBandwidthDI	Integer		Downlink channel bandwidth. Takes one of the following values in MHz • 3 • 5 • 10 • 15 • 20
maximumOutputPower	Float		Maximum power in Watts for the sum of all downlink channels that are allowed to be used simultaneously in a cell.
userCapacity	Integer		Maximum number of pieces of user equipment (UEs) that can connect to this eUtranCell simultaneously.
earfcnDl	Integer		E-UTRA Absolute Radio Frequency Channel Number (downlink). An integer value which identifies the downlink carrier frequency of the cell.
earfcnUl	Integer		E-UTRA Absolute Radio Frequency Channel Number (uplink). An integer value which identifies the uplink carrier frequency of the cell.

Table 393. eUtranCell table (continued)			
Column name	**Type**	**Constraints**	**Description**
TAI	64-character string		Tracking area identifier of the cell. This corresponds to the value stored in the NCIM trackingArea table.
operationalState	Enumeration		Operational state of the LTE cell. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the LTE sector. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

eUtranSector

The `eUtranSector` table models a geographic area of radio coverage that is implemented and supported by physical radio equipment. An eUTRAN sector implements one or more eUTRAN cells.

The following table describes the eUtranSector table.

Table 394. eUtranSector table			
Column name	**Type**	**Constraints**	**Description**
entityID	Integer	FOREIGN KEY NOT NULL	The identifier of an eUtranSector location entity from the `entityData` table.
sectorId	255-character string	NOT NULL	Sector identifier.
sectorName	64-character string	NOT NULL	Sector name.
sectorNumber	Integer		Sector number.
emsDistinguishedName	255-character string		Distinguished name by which the sector is known to its element management system (EMS).
frequencyBand	Integer		Frequency band supported by the sector. All cells serviced by the sector must have carrier frequencies falling within this band.
maximumOutputPower	Float		Available sector power in Watts.

Table 394. eUtranSector table (continued)			
Column name	**Type**	**Constraints**	**Description**
operationalState	Enumeration		Operational state of the sector. This field takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the sector. This field takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

frameRelayEndPoint

The frameRelayEndPoint table represents a logical Frame Relay end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

The following table describes the frameRelayEndPoint table.

Table 395. frameRelayEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a Frame Relay endpoint entity from the `entityData` table.
DLCI	32-bit integer		The data link connection identifier for this Frame Relay endpoint.

Related reference

protocolEndPoint
The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

bgpEndPoint
The bgpEndPoint table represents a logical BGP end point and includes relevant BGP data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table

genericCollection

The `genericCollection` table stores information on generic collections of entities.

The following table describes the `genericCollection` table.

Table 396. genericCollection table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a generic collection from the `entityData` table.

genericRange

The genericRange table holds information about which Customer VLANS are switched through each Virtual Switch Instance (VSI).

The following table describes the genericRange table.

Table 397. genericRange table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	Foreign key Primary key Not null	The entity ID of the interface that is associated with the Customer VLAN range data.
minimumvalue	Integer	Not null	The minimum value of the range. Customer VLANs within the range are switched through the VSI; Customer VLANs outside the range are discarded.
maximumvalue	Integer	Not null	The maximum value of the range.
rangeType	Enumerated	Not null	Can only take the value `CVlanRange`.

geographicLocation

The `geographicLocation` table stores geographic location information for network entities. It does this by creating collections of entities at a specific location.

The following table describes the `geographicLocation` table.

Table 398. geographicLocation table			
Column name	**Type**	**Constraints**	**Description**
altitude	Integer		Vertical height above World Geodetic System WGS84 datum surface (EGM96) at the particular geographical location.

Table 398. geographicLocation table (continued)

Column name	Type	Constraints	Description
altitudeUnits	11-character string		Units to use for the altitude. • 0 – Meters • 1 – Kilometers • 2 – Centimeters • 3 - Feet • 4 – Yards • 5 - Miles • 6 - Inches
entityId	Integer	NOT NULL	The identifier of a geographical location entity from the `entityData` table.
latitude	Fixed-point decimal (up to 10 digits with 8 to the right of the decimal point)		Measurement of latitude for this entity. This is the angular distance (north and south) from the equator on the earth's surface; for example, 78.838753. The value of this attribute is determined based on World Geodetic System WGS84 standard.
locationId	64-character string	NOT NULL	The location identifier by which the location is know to the end-user.
longitude	Fixed-point decimal (up to 11 digits with 8 to the right of the decimal point)		Measurement of longitude for this entity. This is the angular distance (east and west) from the prime meridian on the earth's surface; for example, 35.832636. The value of this attribute is determined based on World Geodetic System WGS84 standard.
locationDescription	255-character string		Free-format textual description of location
timezoneOffset	9-character string		Offset of local time from UTC in format UTC-HH:MM or UTC +HH:MM; for example, UTC +10:30, UTC-6:00.

geographicRegion

The `geographicRegion` table stores geographic region information for radio access network entities. The `geographicRegion` table collects geographic locations from the `geographicLocation` table or other geographic regions from the current table to build a geographical hierarchy.

The following table describes the `geographicRegion` table.

Table 399. geographicRegion table

Column name	Type	Constraints	Description
entityId	Integer	NOT NULL	The identifier of a geographical region from the `entityData` table.
hierachyName	64-character string		The name of hierarchy which this region is part of.
levelName	64-character string		The name of the hierarchy level which this region represents.
hierarchyLevel	Integer		Represents the level of this region within the geographical hierarchy, where the value 1 is the first or lowest level.

globalVlan

The globalVlan table models global VLAN logical collections. A *global VLAN* is a collection of VLAN entities across multiple chassis devices that combine to form a virtual network.

The following table describes the globalVlan table.

Table 400. globalVlan table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of a global VLAN entity from the `entityData` table.
subnet	16-character string	Not null	The subnet address of the VLAN.
netmask	16-character string	Not null	The netmask of the subnet for this VLAN.
vlanClass	Enumerated		The class of the VLAN. Possible values are: `cvlan` (Customer VLAN using QinQ), `svlan` (Service VLAN using QinQ), or `local` (VLAN not using QinQ).
vlanDescr	255-character string		A description of this VLAN.

gnbFunction

The `gnbFunction` table models the role of the gNodeB entity within a network hardware node. Multiple gnbFunction instances may be implemented within a single network hardware node. The gNodeB entity

manages radio air interface communication with users of the 5G network. Each gNodeB controls one or more cells which are geographic areas of radio coverage.

There are three variants of the gnbFunctions will be available for a single GNodeB device. They are listed as follows:

- GNBDU
- GNBCUCP
- GNBCUUP

The following table describes the gnbFunction table.

Table 401. gnbFunction table

Column name	Type	Constraints	Description
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an gnbFunction entity from the `entityData` table.
gNodeBId	64-character string	NOT NULL	Identifier of the gNodeB.
gNodeBIdLength	Integer		
gNodeBName	64-character string	NOT NULL	User friendly name of the gNodeB.
emsDistinguishedName	255-character string		Distinguished name by which the gnbFunction is known to its element management system (EMS).
emsIpAddress	39-character string		IP address of the element management system.
backHaulConnection	39-character string		IP address of the first hop backhaul device to which the gnbFunction is connected; for example, the IP address of a cell-site router.
supportedPLMNs	Integer		Count of the number of PLMNs (Public Land Mobile Networks) supported by the gnbFunction.
MCC	3-character string		An gNodeB can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the gNodeB. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the gNodeB. The MCC consists of three digits.

Table 401. gnbFunction table (continued)

Column name	Type	Constraints	Description
MNC	3-character string		An gNodeB can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the gNodeB. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the gNodeB. The length of the MNC (two or three digits) depends upon the value of the MCC.
vendorName	64-character string		Vendor or manufacturer of the gnbFunction.
gnbFunctionType	10-character string		The gNodeB device manages the radio air interface communication with users of the 5G network. Each gNodeB device controls one or more cells, which are geographic areas of radio coverage. The role of the gNodeB is implemented within a network hardware node and is modelled by NCIM using the gnbFunction entity type. Multiple gnbFunction instances may be implemented within a single network hardware node.
vendorModuleType	64-character string		Vendor-specific gNodeB type.
softwareVersion	64-character string		Vendor-specific gNodeB software version.
userCapacity	Integer		Maximum number of active pieces of user equipment (UEs) that can connect to this gNodeB simultaneously.
maximumOutputPower	Float		Maximum output power of the gNodeB.
operationalState	10-character string		Operational state of the gnbFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown

Table 401. gnbFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
administrativeState	10-character string		Administrative state of the gnbFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

hsrpGroup

The hsrpGroup table represents a Cisco Hot Standby Routing Protocol (HRSP) group logical collection.

The HRSP implements a virtual router with its own IP and MAC addresses. This virtual router forms an HSRP group that consists of a number of real interfaces, only one of which is active at any given time. The active interface forwards IP traffic that is sent to the virtual router and the other interfaces in the group stand by ready to become active if the active interface fails.

The following table describes the hsrpGroup table.

Table 402. hsrpGroup table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a HSRP group entity from the `entityData` table.
virtualIP	32-bit integer	Foreign key Not null	The virtual IP address used by this HSRP group.

hssFunction

The `hssFunction` table models the Home Subscriber Server (HSS). The HSS manages subscriber identities, service profiles, authentication, authorization, and quality of service (QoS), and acts as the master repository for subscriber profiles, device profiles and state information.

The following table describes the hssFunction table.

Table 403. hssFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a hssFunction entity from the `entityData` table.
hssFunctionName	varchar(64)	NOT NULL	Name of the hssFunction (Home Subscriber Server) instance configured on the physical node that implements the hssFunction

Table 403. hssFunction table (continued)

Column name	Type	Constraints	Description
MCC	varchar(3)		An HSS can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the HSS. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the HSS. The MCC consists of three digits.
MNC	varchar(3)		An HSS can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the HSS. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the HSS. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		This is a count of the number of PLMNs (Public Land Mobile Networks) supported by the HSS.
emsDistinguishedName	varchar(255)		The distinguished name by which the HSS is known to its element management system (EMS)
emsIpAddress	varchar(39)		The IP address of the element management system
vendorName	varchar(64)		The vendor/manufacturer of the HSS
vendorModuleType	varchar(64)		Vendor specific HSS Type
softwareVersion	varchar(64)		Vendor specific HSS sofware version
operationalState	Enumeration		The operational state of the hssFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown

Table 403. hssFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
administrativeState	Enumeration		The administrative state of the hssFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

igmpEndPoint

The igmpEndPoint table holds information on the Internet Group Membership Protocol (IGMP) End Points.

Each End Point holds various attributes describing the interface that implements it. There is a dependency between the end point and service associated with it (modeled via the existing dependency table). The End Point is associated with the interface which implements it using the existing protocolEndPoint table.

The following table describes the igmpEndPoint table.

Table 404. igmpEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IGMP End Point.
groups	32-bit integer		A count of the number Cache Table entries for this end point
lastMembQueryIntvl	32-bit integer		Holds the Last Member Query Interval
numJoins	32-bit integer		Number of IGMP group membership joins that have occurred
proxyIfIndex	32-bit integer		Where IGMP proxying is used, this field holds the interface index to which IGMP Host Membership Reports are sent.
querier	25 character string		IP address of the IGMP Querier
querierExpiryTime	64-bit integer		Remaining time before expiry of the Other Querier Present Timer expires. (0 if local system is the querier)
querierUpTime	64-bit integer		Time ticks since the Querier last changed

Table 404. igmpEndPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
queryInterval	32-bit integer		How often IGMP query messages are sent on the interface associated with this End Point.
queryMaxResponse Time	32-bit integer		Maximum response time in tenths of a second
robustness	32-bit integer		The Robustness Variable used to alter IGMP sensitivity to packet loss
status	Enumerated Value	Takes one of the following values: • ‘other' • ‘unknown' • ‘enabled' • ‘disabled'	The status of IGMP on the interface associated with the End Point.
version	32-bit integer		Version of IGMP running
versionQuerierTimer	64-bit integer		Remaining time before it is assumed that no IGMP routers are present on the interface associated with the End Point
wrongVersionQueries	32-bit integer		A count of number of queries received with unexpected IGMP versions. A non-zero value indicates an IGMP configuration issue.

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

igmpGroup

The igmpGroup table holds multicast group collections for which there are associated Internet Group Membership Protocol (IGMP) end points in the igmpEndPoint table. Entries in the igmpGroup table collect associated igmpEndPoints using the existing collects table.

The following table describes the igmpGroup table.

Table 405. igmpGroup table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IGMP group

Table 405. igmpGroup table (continued)			
Column name	**Type**	**Constraints**	**Description**
groupAddress	25-character string		IP address of the multicast group
groupMask	25-character string		Mask of the multicast group
groupName	60-character string		Name associated with the group

igmpService

The igmpService table represents an Internet Group Management Protocol (IGMP) service. Each row in the table corresponds to a single hosted IGMP service. The service is associated with the device on which it runs using the hostedService table.

The following table describes the igmpService table.

Table 406. igmpService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IGMP Service

ipConnection

The `ipConnection` table stores information on IP connections.

The following table describes the `ipConnection` table.

Table 407. ipConnection table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of an IP connection from the `entityData` table.

ipEndPoint

The ipEndPoint table represents an IP end point and includes relevant data. The endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

The following table describes the ipEndPoint table:

Table 408. ipEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32–bit integer	Foreign key Not null	The identifier of an IP end point entity from the `entityData` table.
DNSName	255–character string		DNS name for the IP address associated with this IP end point.
address	39–character string	Not null	IP address associated with this IP end point.

Table 408. ipEndPoint table (continued)

Column name	Type	Constraints	Description
ipNumber	64–bit integer, unsigned	Not null	For IPv4: IP address represented as a 32–bit integer. For IPv6: The first half of the IP address is represented as a 128–bit integer. The value is shown as a signed 64-bit integer. For example, for the IPv6 address fe80:0000:0000:0000:21a:30ff:fe2b:fb80, the hexadecimal ipNumber is FE80 0000 0000 0000.
netNumber	64–bit integer	Not null	For IPv4: Not applicable For IPv6: Second half of IP address represented as a 128–bit integer. The value is shown as a signed 64-bit integer. For example, for the IPv6 address fe80:0000:0000:0000:21a:30ff:fe2b:fb80, the hexadecimal netNumber is 021A 30FF FE2B FB80. (The decimal number is 18338657682652659712 as an unsigned integer and 108086391056891904 as a signed integer.)
subnet	39–character string		Subnet to which the IP address belongs.
netmask	39–character string		Netmask for the subnet.
netmaskbits	32–bit integer		Netmask bits for the subnet
addressSpace	255–character string		Relevant NAT address space if network address translation is being used.
protocol	String value		An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
cdmAddressType	16-bit integer	Not null Default value: 0	The IBM Common Data Model defines an attribute named AddressType in the IpV4Address and IPv6Address views and the ipEndPoint class is the equivalent object in NCIM. Therefore to make the attributes in the classes consistent, a new attribute named cdmAddressType has been added to the ipEndPoint table.

Related concepts
Technology-specific data

NCIM models a range of different network technologies, including IP, VLANs, and MPLS VLANs.

Related reference

protocolEndPoint
The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

bgpEndPoint
The bgpEndPoint table represents a logical BGP end point and includes relevant BGP data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table

ipMRouteDownstream

The ipMRouteDownstream table holds the downstream route statistics per device or MDT. Each entity in this table will have a dependency relationship with the associated MDT via the existing dependency table.

The following table describes the ipMRouteDownstream table.

Table 409. ipMRouteDownstream table

Column name	Type	Constraints	Description
closestMemberHops	32-bit integer		The number of hops that it will take to reach the closest member of this multicast group.
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IGMP End Point
expiryTime	32-bit integer		Time ticks left before the route entry expires
hopState	Enumerated value	Takes one of the following values: • unknown • other • pruned • forwarding	Indicates whether the downstream interface is being used to forward packets.
nextHop	15-character string		IP Address of the next downstream hop. This address may be the multicast group address.
outInterface	32-bit integer		NULL if ifIndex was 0
packetCount	32-bit integer		The number of packets from sources destined for the group.

Table 409. ipMRouteDownstream table (continued)			
Column name	**Type**	**Constraints**	**Description**
protocol	Enumerated value	Takes one of the following values: • other • local • netmgmt • DVMRP • MOSPF • PIMSparseDense • CBT • PIMSparseMode • PIMDenseMode • IGMPOnly • BGMP • MSDP	The protocol used to learn the downstream next hop route
startTime	Timestamp		Approximate time that the route entry was learnt; calculated using uptime and the system time at the time the device was interrogated.
uptime	32-bit integer		Time ticks since this route entry was learnt

ipMRouteEndPoint

The ipMRouteEndPoint table holds information on the IP Multicast Routing Protocol End Points.

Each End Point holds various attributes describing the interface that implements it. There is a dependency between the end point and service associated with it (modelled using the existing dependency table). The End Point is associated with the interface which implements it through the existing protocolEndPoint table.

The following table describes the ipMRouteEndPoint table.

Table 410. ipMRouteEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IP Multicast Routing Protocol End Point
flowDirection	Enumerated value	Takes one of the following values: • upstream • downstream	Indicates the direction of flow represented by this end point

Table 410. ipMRouteEndPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
isLastHop	32-bit integer	0 or 1	Indicates whether this end point represents a last known hop
protocol	Enumerated value	Takes one of the following values: • other • local • netmgmt • DVMRP • MOSPF • PIMSparseDense • CBT • PIMSparseMode • PIMDenseMode • IGMPOnly • BGMP • MSDP	Routing protocol running on the interface associated with this End Point
rateLimit	32-bit integer		Rate limit in kbps of multicast traffic on the interface associated with this End Point
ttl	32-bit integer		Time To Live counter

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

ipMRouteGroup

The ipMRouteGroup table represents Multicast Groups, as contained by the Multicast Distribution Tree (MDT).

The following table describes the ipMRouteGroup table.

Table 411. ipMRouteGroup table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IP Multicast Routing Group
groupAddress	15-character string	Not Null	The address of the multicast group
groupMask	15-character string		15-character string

ipMRouteMdt

The ipMRouteMdt table holds the Collection entities representing the Multicast Distribution Trees (MDTs) for each Multicast Source/Group. The name of the entity (present in the `entityData` table) takes the form (S,G). Each row in the table represents a single MDT. The MDT collects the End Points involved using the existing collects table. The MDT contains their related Groups/Source entities (through the existing contains table), and depend on the Multicast Route entities (through the dependency table).

The following table describes the ipMRouteMdt table.

Table 412. ipMRouteMdt table

Column name	Type	Constraints	Description
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IP Multicast Routing MDT entity
mdtType	Enumerated value	Takes one of the following values: • unknown • other • SPT • SDT	Holds the type of MDT represented

ipMRouteService

The ipMRouteService table represents an IP Multicast Routing service and includes any attributes relevant to the service. Each row in the table corresponds to a single hosted IPMRouting service. The service is associated with the device on which it runs through the hostedService table.

The following table describes the ipMRouteService table.

Table 413. ipMRouteService table

Column name	Type	Constraints	Description
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IP Multicast Routing Service
routeCount	32-bit integer		The number of routes known to this Service

ipMRouteSource

The ipMRouteSource table represents Multicast Sources, as contained by the Multicast Distribution Tree (MDT).

The following table describes the ipMRouteSource table.

Table 414. ipMRouteSource table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the IP Multicast Routing Source
source	15-character string	Not Null	The address of the source
sourceMask	15-character string		The mask of the source

ipMRouteUpstream

The ipMRouteUpstream table holds the upstream (RPF) route statistics for each device or MDT. Each entity in this table has a dependency relationship with the associated MDT through the existing dependency table.

The following table describes the ipMRouteUpstream table.

Table 415. ipMRouteUpstream table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	• Primary Key • Foreign Key (entityData table) • Not Null	The identifier of the RPF upstream route entity
inInterface	32-bit integer		NULL if ifIndex was 0
upstreamNbr	15-character string		IP address of the RPF neighbour (if known)
uptime	32-bit integer		Time ticks since this route entry was learnt
expiryTime	32-bit integer		Time ticks left before the route entry expires
startTime	Timestamp		Approximate time that the route entry was learnt; calculated using uptime and the system time at the time the device was interrogated.
packetCount	32-bit integer		The number of packets from sources destined for the group.
differentInIfPackets	32-bit integer		The number of packets dropped because they were received on the wrong interface
octets	32-bit integer		The number of octets forwarded

Table 415. ipMRouteUpstream table (continued)

Column name	Type	Constraints	Description
protocol	Enumerated value	Takes one of the following values: • other • local • netmgmt • DVMRP • MOSPF • PIMSparseDense • CBT • PIMSparseMode • PIMDenseMode • IGMPOnly • BGMP • MSDP	The protocol used to learn this route entry
rtProto	Enumerated value	Takes one of the following values; • other • local • netmgmt • ICMP • EGP • GGP • hello • RIP • ISIS • ESIS • ciscoIGRP • bbnSpfIGP • OSPF • BGP • IDRP • ciscoEIGRP • DVMRP	The protocol used to learn the upstream interface for this route entry
rtAddress	15-character string		Address used to determine the upstream interface
rtMask	15-character string		Mask used to determine the upstream interface

Table 415. ipMRouteUpstream table (continued)			
Column name	**Type**	**Constraints**	**Description**
rtType	Enumerated Value	Takes one of the following values: • unknown • other • unicast • multicast	The type of route

ipPath

The `ipPath` table stores information on IP paths.

The following table describes the `ipPath` table.

Table 416. ipPath table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	Not null	The identifier of an IP pathfrom the `entityData` table.
fromIP	39-character string		Starting IP address for this path.
toIP	39-character string		End IP address for this path.
viaIP	39-character string		IP address that this path must traverse.
hops	32-bit integer	Not null	Number of hops in the path.
ecmp			Indicates whether equal-cost multi-path routing applies to this path.
asymmetric			Indicates whether this is an asymmetric path.
protocols	100-character string		Indicates the protocols used in this path.
cost	32-bit integer		

itnmService

The itnmService table represents Service-Affected Events (SAE); the table is used by the SAE plug-ins in the Event Gateway, ncp_g_event. This table is used only indirectly by the SAE plug-ins, since the SAE plug-ins use the NCIM cache tables, which are based on NCIM tables.

The following table describes the itnmService table.

Table 417. itnmService table			
Column name	**Type**	**Constraints**	**Description**
entityID	32–bit integer	Not null	The identifier of the SAE entity from the `entityData` table.
serviceName	255–character string	Not null	The name of the SAE.
serviceType	64–character string	Not null	The type of SAE

lagEndPoint

The lagEndPoint table holds information about end points within Link Aggregation Groups (LAGs).

The following table describes the lagEndPoint table.

Table 418. lagEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Not null Primary Key Foreign Key	Automatically incremented ID that provides a unique value for each end point across all domains.
lagId	32-bit integer	Not null	The ID of the LAG.
priority	32-bit integer		The priority of the end point.
lagMode	8-character string		The mode that the LAG is operating in. Takes one of the following values: • lacp • static • unknown • other • disabled
actorAdminState	255-character string		The administrative value of Actor_State as transmitted by the actor in the Link Aggregation Control PDUs.
actorOperState	255-character string		The operational value of Actor_State as transmitted by the actor in the Link Aggregation Control PDUs.
actorSystemId	255-character string		The MAC address value used as a unique identifier for the server that contains this LAG.
actorAdminKey	Integer		Administrative value of the key for the LAG.
actorOperKey	Integer		Operational value of the key for the LAG.

Table 418. lagEndPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
partnerAdminState	255-character string		The administrative value of Actor_State for the protocol partner.
partnerOperState	255-character string		The value of Actor_State in the most recently received Link Aggregation Control PDU transmitted by the protocol partner.
partnerSystemId	255-character string		The MAC address value used as a unique identifier for the current protocol partner of this LAG.
partnerAdminKey	Integer		Administrative value of the key for the protocol partner.
partnerOperKey	Integer		Operational value of the key for the protocol partner.

lingerTime

The lingerTime table stores the linger time for a device. The linger time is the number of discoveries that a device can fail to be found in before it is removed from the topology.

The linger time is set for a device when it is instantiated, from the default value in the model.config table. Each time that a device in the topology is not discovered, the linger time is decreased by 1. When the linger time is zero, if the device is not discovered, it is removed from the topology.

The lingerTime table is described in the following table:

Table 419. lingerTime table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a device.
lingerTime	Integer	Not null	The current linger time of the device.

Related reference

ncimCache.lingerTime table

The ncimCache.lingerTime table stores the linger time for a device.

localVlan

The localVlan table specifies which global VLAN the *local VLAN* belongs to. A local VLAN represents all the interfaces on a single chassis device that belong to a global VLAN.

In addition, the `contains` table specifies the interfaces contained by the local VLAN.

Table 420. localVlan Ttable			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a local VLAN entity from the `entityData` table.
vlanClass	Enumerated		The class of the VLAN. Possible values are: `cvlan` (Customer VLAN using QinQ), `svlan` (Service VLAN using QinQ), or `local` (VLAN not using QinQ).
vlanId	32-bit integer	Not null	The identifier of a connection from the `connects` table.
vlanDescr	255-character string		The description of a connection from the `connects` table.
vlanName	64-character string		The name of a connection from the `connects` table.
vlanType	32-character string		The type of a connection from the `connects` table.
vlanState	32-character string		The state of a connection from the `connects` table.

Related reference

contains

The `contains` table stores data on physical and logical containment. This table belongs to the category *containment*.

lteInterface

The `lteInterface` table models the different types of interfaces between LTE devices.

LTE interfaces modelled in NCIM

The following table lists the LTE interface types that are modelled in NCIM, and shows which interface types are used to connect different LTE elements. The interface types are described below the table.

Table 421. LTE interfaces modelled in NCIM								
LTE element	**EIR**	**eNode B**	**HSS**	**MME**	**PCRF**	**PGW**	**SGW**	**SGSN**
Equipment Identity Register (EIR) “eirFunction” on page 649				S13				
Evolved NodeB “enbFunction” on page 651		X2		S1-MME			S1-U	
Home Subscriber Server (HSS) “hssFunction” on page 662				S6a				
Mobility Management Entity (MME) “mmeFunction” on page 681	S13	S1-MME	S6a	S10			S11	S3

Table 421. LTE interfaces modelled in NCIM (continued)								
LTE element	**EIR**	**eNode B**	**HSS**	**MME**	**PCRF**	**PGW**	**SGW**	**SGSN**
Policy and Charging Rules Function (PCRF) "pcrfFunction" on page 702						Gx		
Packet Data Network Gateway (PGW) "pgwFunction" on page 704					Gx		S5 S8	
Serving Gateway (SGW) "sgwFunction" on page 744		S1-U		S11		S5 S8		S4
Serving GPRS Support Node (SGSN) "ranSGSN" on page 742							S4	

Gx
Interface between the PGW and the Policy and Charging Rules Function (PCRF). In particular, this is the interface between the Policy Control Enforcement Function (PCEF) or the PGW and the PCRF.

OAM
Operational and maintenance interface, used to manage the device.

SGi
Interface between the PGW and any packet data network.

S1
Interface between the eNodeB and the core network.

S1-MME
Control plane interface between an eNodeB and an MME.

S1-U
User plane interface between an eNodeB and one or more SGWs.

S3
Control plane interface between an MME and a Serving GPRS Support Node (SGSN). The interface is used to manage mobility inter-3GPP Radio Access Technology (RAT) mobility between, for example, LTE and UMTS or GPRS.

S4
Control and user plane interface between an SGW and a Serving GPRS Support Node (SGSN).

S5
Modelled in NCIM as a user plane interface between an SGW and a PGW within the same Public Land Mobile Network (PLMN).

Note: The S5 interface can carry control and user plane data but in NCIM it is modelled as user plane only.

S6a
Control plane interface between MME and Home Subscriber Server (HSS).

S8
Equivalent to the S5 interface except that it connects an SGW in the Visited PLMN (VPLMN) to a PGW in the roaming subscriber's Home PLMN (HPLMN).

Note: The S5 interface can carry control and user plane data but in NCIM it is modelled as user plane only.

S10
Inter-MME control plane interface.

S11
Control plane interface between an MME and an SGW.

S13
Control plane interface between MME and the Equipment Identity Register (EIR).

X2
Modelled in NCIM as a control plane interface between neighbouring eNodeBs. Used for signalling between neighbouring eNodeBs.

Note: The X2 interface can carry control and user plane data but in NCIM it is modelled as control plane only.

The following table describes the lteInterface table.

Table 422. lteInterface table

Column name	Type	Constraints	Description
entityId	Integer	FOREIGN KEY NOT NULL	
interfaceType	Enumeration		Type of LTE interface. Takes one of the following values: • Gx • OAM • SGi • S1 • S1-MME • S1-U • S3 • S4 • S5 • S6a • S8 • S10 • S11 • S13 • X2

ltePool

The `ltePool` table is a generic modelling mechanism for groups of pooled LTE entities, and is used to model MME pools, PGW pools, and SGW pools. As an example, in order to model an MME pool, the relationship between the ltePool entity and associated mmeFunction entities is modelled using the collects table.

The following table describes the ltePool table.

Table 423. ltePool table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an ltePool entity from the `entityData` table.
lteGroupId	64-character string		The group identifier of the LTE pool.
ltePoolName	64-character string	NOT NULL	The name of the MME Pool.
ltePoolType	Enumeration		Type of LTE pool. Takes one of the following values: • MME • SGW • PGW
ltePoolArea	64-character string		The name of the Pool Area that is covered by the pool. There is a one to one mapping between a Pool and a Pool Area.

managedStatus

The `managedStatus` table stores the managed status information for each network entity in the topology.

The following table describes the `managedStatus` table.

Table 424. managedStatus table			
Column name	**Type**	**Constraints**	**Description**
entityId	32–bit integer	Foreign key Not null	Identifier of an entity from the entityData table.

Table 424. managedStatus table (continued)

Column name	Type	Constraints	Description
status	8–bit integer		The managed status of an entity can be one of the following values: **0** Managed state. The entity is managed. A device can be set to managed by using the Topoviz or the Structure Browser GUIs, or by using the `ManagedNode.pl` or `RemoveNode.pl` scripts. **1** Unmanaged state. The entity is unmanaged. A device can be set to unmanaged by using the Topoviz or the Structure Browser GUIs, or by using the `UnManagedNode.pl` or `RemoveNode.pl` scripts. **2** Unmanaged by ncp_disco. This setting cannot be modified from the GUI. This value is set by the `PopulateDNCIM_ManagedStatus.stch` stitcher. **3** Unmanaged because the IP address is out of the discovery scope. The device has been discovered through another IP address that is within the discovery scope. A managed status of 3 is usually given to interfaces, rather than chassis. This value is set by the `PopulateDNCIM_ManagedStatus.stch` stitcher. **Note:** Unmanaged entities do not suppress other events in RCA. The ncp_poller process does not poll unmanaged entities. Events on unmanaged entities have the field NmosManagedStatus set in the alerts.status field in the ObjectServer.
username	240–character string		Name of the user who last set the status of the entity.
changeTime	Timestamp		Date and time when the status of the entity was changed.
maintenance StartTime	Timestamp		Start time for device to be in unmanaged state.
maintenance EndTime	Timestamp		End time for device to be in unmanaged state.

Related reference

ncimCache.managedStatus table

The ncimCache.managedStatus table stores the managed status information for network entities.

mmeFunction

The `mmeFunction` table models the role of the Mobility Management Entity (MME) within a network hardware node. Multiple mmeFunction instances can be implemented within a single network hardware node. The MME is the main signalling control element in the core network and is the key control node for enabling user equipment access to the core network.

The following table describes the mmeFunction table.

Table 425. mmeFunction table

Column name	Type	Constraints	Description
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an mmeFunction location entity from the `entityData` table.
MMEC	64-character string		MME code. Uniquely identifies an MME within an MME Group. This attribute consists of 8 bits.
MMEGI	64-character string		MME group identifier. Uniquely identifies an MME Group within a public land mobile network (PLMN). This attribute consists of 16 bits.
mmeName	64-character string	NOT NULL	Uniquely identifies an MME when more than one MME configured on same device.
MCC	3-character string		An MME can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the MME. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the MME. The MCC consists of three digits.
MNC	3-character string		An MME can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the MME. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the MME. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		Count of the number of PLMNs supported by the MME.
emsDistinguishedName	255-character string		Distinguished name by which the MME is known to its element management system (EMS).
emsIpAddress	39-character string		IP address of the element management system.
vendorName	64-character string		Vendor or manufacturer of the MME.
vendorModuleType	64-character string		Vendor-specific MME type.
softwareVersion	64-character string		Vendor-specific MME sofware version.
operationalState	Enumeration		Operational state of the mmeFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown

Table 425. mmeFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
administrativeState	Enumeration		Administrative state of the mmeFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

mplsTEService

The mplsTEService table represents an MPLS TE service and includes relevant protocol data. This MPLS TE service runs on a device, as modeled in the hostedService table. Each row in this table corresponds to a single hosted MPLS TE service. MPLS TE configured devices will host only one MPLS TE service, which in turn can support multiple MPLS TE tunnels.

The following table describes the mplsTEService table.

Table 426. mplsTEService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign Key Not Null	The identifier of an MPLS TE service from the `entityData` table.
numTunnelsConfigured	32-bit integer		The number of configured tunnels represented by this service.
numTunnelsActive	32-bit integer		The number of active tunnels represented by this service.
teDistributionProtos	30-character string		Names of the TE distribution protocols in use by the service.
maxNumTunnelHops	32-bit integer		Maximum number of hops a TE tunnel is allowed to make.

mplsTETunnel

The mplsTETunnel table represents the MPLS TE tunnels discovered in the network and includes a number of tunnel attributes. Each row in this table corresponds to a TE tunnel discovered on a device.

The following table describes the mplsTETunnel table.

Table 427. mplsTETunnel table			
Column name	**Type**	**Constraints**	**Description**
adminStatus	Enumerated value	Takes one of the following values: • up • down • testing	Administrative status

Table 427. mplsTETunnel table (continued)

Column name	Type	Constraints	Description
creationTime	Timestamp		Time when tunnel was created
description	100-character string		The description of the tunnel
egressLSRId	15-character string		Egress LSR ID of the tunnel
entityId	32-bit integer	Foreign Key Not Null	The identifier of an MPLS TE service from the `entityData` table
excludeAllAffinity	32-bit integer		Exclude-All constraint
holdPriority	32-bit integer		Hold priority
includeAllAffinity	32-bit integer		Include-All constraint
includeAnyAffinity	32-bit integer		Include-Any constraint
ingressLSRId	15-character string		Ingress LSR ID of the tunnel
instanceId	25-character string		Unique tunnel identifier
instancePriority	32-bit integer		Priority of this tunnel instance
locallyProtected	8-bit integer		Denotes whether tunnel is locally protected or not
name	50-character string		The name of the tunnel
numPathChanges	32-bit integer		Number of times the tunnel path has changed
operStatus	Enumerated value	Takes one of the following values: • up • down • testing • unknown • dormant • notPresent • lowerLayer Down	Operational status
owner	20-character string		Owner of the tunnel
resourcePointer	32-bit integer		Index into the resource table for this tunnel.
role	Enumerated value	Takes one of the following values: • head • transit • tail	Role of this tunnel
sessionAttributes	90-character string		Tunnel session attributes in text form

Table 427. mplsTETunnel table (continued)			
Column name	**Type**	**Constraints**	**Description**
setupPriority	32-bit integer		Setup priority
signallingProtocol	Enumerated value	Takes one of the following values: • none • rsvp • crldp • other	The protocol used to signal the tunnel
transitions	32-bit integer		Number of times the state has changed
tunnelInstance	25-character string		Identifies a specific instance of the tunnel identified by tunnelIndex
tunnelIndex	25-character string		Tunnel index
uptime	32-bit integer		Amount of time tunnel has been up
xcPointer	128-character string		Index into the LSP cross-connect table for this tunnel

mplsTETunnelEndPoint

The mplsTETunnelEndPoint table represents an MPLS TE protocol end point and is implemented on the interface associated with the configured tunnel. The end point references the associated TE tunnels unique instance id.

The following table describes the mplsTETunnelEndPoint table.

Table 428. mplsTETunnelEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign Key Not Null	The identifier of an MPLS TE service from the `entityData` table
instanceId	25-character string		Unique tunnel identifier, as seen in the mplsTETunnel table

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

mplsTETunnelResource

The mplsTETunnelResource table represents the MPLS TE Tunnel resource configurations that tunnels might be associated with.

The following table describes the mplsTETunnelResource table.

Table 429. mplsTETunnelResource table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign Key Not Null	The identifier of an MPLS TE service from the `entityData` table.
resourceIndex	32-bit integer		Resource index. This is the value associated with the mplsTETunnel tables resourcePointer.
maxRate	32-bit integer		Maximum data rate in bits per second, where 0 means best effort.
meanRate	32-bit integer		Average data rate in bits per second. 0 means best effort
maxBurstSize	32-bit integer		Maximum burst size in bytes
meanBurstSize	32-bit integer		Average burst size in bytes

mplsLSP

The mplsLSP table represents LSPs (Label Switched Paths) that might be traversed by MPLS TE tunnels.

The following table describes the mplsLSP table.

Table 430. mplsLSP table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign Key Not Null	The identifier of an MPLS TE service from the `entityData` table.
lspID	40-character string		The ID of the LSP.

multiplexer

The `multiplexer` table describes multiplexer entities.

The following table describes the `multiplexer` table.

Table 431. multiplexer table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a multiplexer entity from the `entityData` table.

netcoolAsmsRunning

The netcoolAsmsRunning table lists instances of ASM running on main node devices.

The following table describes the netcoolAsmsRunning table.

Table 432. netcoolAsmsRunning table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a chassis device from the `entityData` table.
ASMName	64-character string	Not null	The name of an ASM running on this chassis device.

networkInterface

The `networkInterface` table represents interfaces on a chassis device.

The following table describes the networkInterface table.

Table 433. networkInterface			
Column name	**Type**	**Constraints**	**Description**
entityId	32–bit integer	Foreign key Not null	The identifier of a physical interface from the `entityData` table.
ifTypeString	45-character string		The textual string for the interface type.
aid	255-character string		TL1 access identifier.
alternativeName	255-character string		Alternative name for the device.
bandwidth	NUMBER(19)		Bandwidth of the interface measured in kilobits per second.
configuredDuplex	Enumerated value		Current® administrative duplex setting for this interface. Takes one of the following values: • HalfDuplex • FullDuplex • Auto • Unknown • Other
connectorPresent	Enumerated value		Indicates whether the interface has a connector. Takes one of the following values: • True • False
dataLinkLayer Discovery	Integer		An indication as to whether this interface is running a data-link-layer discovery protocol.

Table 433. networkInterface (continued)

Column name	Type	Constraints	Description
encapsulation	Tiny integer		Current encapsulation used on the interface.
ifType	32–bit integer		The interface type.
keepalive	32–bit integer		Shows the duration between keepalive messages.
mediaType	Tiny integer		Indicates the physical media being used by this interface.
mtu	32–bit integer		The maximum transmission unit for this interface.
name	255-character string		Name of the interface.
operationalDuplex	Enumerated value		Actual duplex of the interface. Takes one of the following values: • HalfDuplex • FullDuplex • Auto • Unknown • Other
operationalStatus	12-character string		Takes one of the following values: • unknown • other • started • stopped
physicalAddress	255–character string		The physical address of the interface.
promiscuous	5–character string		Indicates whether this interface only accepts packets or frames addressed to this station. Takes one of the following values: • True • False

Table 433. networkInterface (continued)

Column name	Type	Constraints	Description
sendLinkStateAlarm	5–character string		Indicates whether the interface or port is configured to send link state alarms to a management station. Takes one of the following values: • True • False
ifIndex	32–bit integer		The index of the interface.
speed	64–bit integer		Normalized actual available speed of the interface measured in bits per second.
switchPortMode	Tiny integer		Indicates whether this physical interface is a VLAN trunk port.
ifName	128-character string		The name assigned to the interface.
ifDescr	255-character string		A description of the interface.
ifAlias	255-character string		The alias for the interface.
ifSpeed	64–bit integer		An estimate of the current bandwidth of the interface in bits per second.
ifHighSpeed	32–bit integer		An estimate of the current bandwidth of the interface in units of 1,000,000 bits per second.
ifAdminStatus	Enumerated value		The required state of the interface. Takes one of the following values: • Up • Down • Testing

Table 433. networkInterface (continued)			
Column name	**Type**	**Constraints**	**Description**
ifOperStatus	Enumerated value		The current operational state of the interface. Takes one of the following values: • Up • Down • Testing • unknown • dormant • notPresent • lowerLayerDown
accessIPAddress	39-character string		The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.
accessProtocol	Enumerated type (string 7 chars)		An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device
portNumber	32–bit integer		The port number for this interface on the chassis device. The method of determining the port number is dependent on the make and model of the device that is discovered. For this reason, use this value with caution.
status	255-character string		Status of this entity.

networkServiceEntityEndPoint

The `networkServiceEntityEndPoint` table describes network service entity endpoints.

The following table describes the `networkServiceEntityEndPoint` table.

Table 434. networkServiceEntityEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a network service entity from the `entityData` table.
NSEI	Integer		Network service entity identifier. An NSEI is used in the management of frame relay links. The `networkServiceEntityEndPoint` object helps model that relationship.

networkVpn

The networkVpn table represents a logical collection of IP addresses collected within a VPN.

The following table describes the networkVpn table.

Table 435. networkVpn table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a network VPN from the `entityData` table.
VPNName	255-character string	Not null	The name of the VPN.
VPNType	64-character string	Not null	The type of VPN.

nrCellCU

The `nrCellCU` table models a geographical area of radio coverage that is implemented and supported by physical radio equipment for 5G NR GNB devices, such as towers, amplifiers, and antennas. These are contained in GNBCUCP/GNBCUUP Functions.

The following table describes the nrCellCU table.

Table 436. nrCellCU table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an nrCellCU entity from the `entityData` table.
nRCellName	64-character string	NOT NULL	Cell identifier or name of the cell.
emsDistinguishedName	255-character string		Distinguished name by which the nrCellCU is known to its element management system (EMS).
nRCellId	64-character string	NOT NULL	Uniquely identifies a cell within a PLMN. It is often constructed from gNodeB ID + Physical Cell ID.

Table 436. nrCellCU table (continued)

Column name	Type	Constraints	Description
nRCellState	10-character string		Represents the active state of the cell. Takes one of the following values: • IDLE • ACTIVE • INACTIVE • UNKNOWN
TAI	64-character string	NOT NULL	Tracking Area Identifier (TAI). This is a globally unique tracking area identifier, made up of the PLMN ID and the TAC.
physicalCellID	Integer		Physical cell identifier. Takes a value in the range 0 to 503. The physical cell id is used by the cell to encode and decode the data that it transmits. It is used in a similar way to the UMTS scrambling code. To avoid interference, neighboring cells should have different physical cell identifiers. The physical cell id is derived from the primary and secondary synchronization signals (PSS and SSS). The PSS takes a value from 0 to 2, the SSS takes a value from 0 to 167, and the physical cell id is determined based on the following formula: `PSS + 3*SSS` The result of this calculation equates to a value of between 0 and 503.
localCellId	Integer		Local cell id unique within the nrCellCU.
operationalState	10-character string		Operational state of the nrCellCU. Takes one of the following values: • Enabled • Disabled • Other • Unknown

Table 436. nrCellCU table (continued)			
Column name	**Type**	**Constraints**	**Description**
administrativeState	13-character string		Administrative state of the nrCellCU. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

nrCellDU

The `nrCellDU` table models a geographical area of radio coverage that is implemented and supported by physical radio equipment for 5G NR GNB devices, such as towers, amplifiers, and antennas. These are contained in GNBDU Functions.

The following table describes the nrCellDU table.

Table 437. nrCellDU table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an nrCellDU entity from the `entityData` table.
nRCellName	64-character string	NOT NULL	Cell identifier or name of the cell.
emsDistinguishedName	255-character string		Distinguished name by which the nrCellDU is known to its element management system (EMS).
nRCellId	64-character string	NOT NULL	Uniquely identifies a cell within a PLMN. It is often constructed from gNodeB ID + Physical Cell ID.
nRCellState	10-character string		Represents the active state of the cell. Takes one of the following values: • IDLE • ACTIVE • INACTIVE • UNKNOWN
TAI	64-character string	NOT NULL	Tracking Area Identifier (TAI). This is a globally unique tracking area identifier, made up of the PLMN ID and the TAC.
channelBandwidthUl	Float		Uplink channel bandwidth.
channelBandwidthDl	Float		Downlink channel bandwidth.

Table 437. nrCellDU table (continued)

Column name	Type	Constraints	Description
maximumOutputPower	Float		Maximum power in Watts for the sum of all downlink channels that are allowed to be used simultaneously in a cell.
userCapacity	Integer		Maximum number of pieces of user equipment (UEs) that can connect to this nrCellDU simultaneously.
physicalCellID	Integer		Physical cell identifier. Takes a value in the range 0 to 503. The physical cell id is used by the cell to encode and decode the data that it transmits. It is used in a similar way to the UMTS scrambling code. To avoid interference, neighboring cells should have different physical cell identifiers. The physical cell id is derived from the primary and secondary synchronization signals (PSS and SSS). The PSS takes a value from 0 to 2, the SSS takes a value from 0 to 167, and the physical cell id is determined based on the following formula: `PSS + 3*SSS` The result of this calculation equates to a value of between 0 and 503.
localCellId	Integer		Local cell id unique within the nrCellDU.
arfcnDl	Integer		Absolute Radio Frequency Channel Number (downlink). An integer value which identifies the downlink carrier frequency of the cell.
arfcnUl	Integer		Absolute Radio Frequency Channel Number (uplink). An integer value which identifies the uplink carrier frequency of the cell.
nRPCI	64-character string		Holds the Physical Cell Identity (PCI) of the NR cell

Table 437. nrCellDU table (continued)

Column name	Type	Constraints	Description
ssbFreq	Float		Indicates cell defining SSB frequency domain position. Frequency of the cell defining SSB transmission. The frequency provided in this attribute identifies the position of resource element. The frequency shall be positioned on the NR global frequency raster, and within bSChannelBwDL. Allowed values: 0..3279165
ssbPeriodicity	Integer		Indicates cell defined SSB periodicity in number of subframes(ms). The SSB periodicity in msec is used for the rate matching purpose. Allowed values: 5, 10, 20, 40, 80, 160
ssbSubCarrierSpacing	Integer		This SSB is used for synchronization. Its units are in kHz. Allowed values: {15, 30, 120, 240} **Note:** The allowed values of SSB used for representing data. For example, a BWP is 15, 30, 60 and 120 in units of kHz.
operationalState	10-character string		Operational state of the nrCellDU. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	13-character string		Administrative state of the nrCellDU. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

operatingSystem

The `operatingSystem` table represents the software responsible for interacting with hardware devices.

The following table describes the `operatingSystem` table.

Table 438. operatingSystem table

Column name	Type	Constraints	Description
entityId	Integer	not null	The identifier of an operating system entity from the `entityData` table.
assetID	255-character string		Asset identifier for this entity.
assetTag	255-character string		Asset tag for this entity.
bootTime	64-bit integer		Time duration the operating system has been online.
buildLevel	255-character string		Build level of the operating system without any version or release information.
ciCategory	255-character string		Configuration item category for this entity.
ciRole	Tiny integer		Identifies the environment, or role, in which a configuration item (CI) resides. For example, if a CI is set aside for test purposes, then this column can be set to a value of `Test`. If a role is needed that is not defined in the enumeration for ciRole, then use the value `Other`.
configLastUpdate	255-character string		UTC date and time when the information was last altered in the source application.
currentTime	64-bit integer		Current time of the instance of the operating system.
fqdn	255-character string		Fully-qualified host name assigned to the operating system. In cases where the datacenter does not implement the Domain Name System (DNS), the fully-qualified host name is the short name.
generalCIRole	255-character string		Environment, or role, in which a CI resides.

Table 438. operatingSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
kernelArchitecture	255-character string		Raw details of the supported system architecture of the kernel component of the operating system.
kernelVersion	255-character string		Raw version of the kernel component of the operating system.
lastAuditState	Tiny integer		Last audit state for this device. Possible values are: • 0 Unknown • 1 Other • 2 Good • 3 No Physical CI • 4 No CMDB Record • 5 Inaccurate CMDB Record
lastAuditTime	Timestamp		Last audit time this entity.
lastLifecycleStateTime	Timestamp		Last lifecycle state time for this entity.
osLevel	Integer		Operating system level.

Table 438. operatingSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
lifecycleState	Tiny integer		Lifecycle state for this device. Possible values are: • 0 Unknown • 1 Other • 2 Ordered • 3 Received • 4 In Test • 5 Tested • 6 Installed • 7 Enabled • 8 Disabled • 9 In Maintenance • 10 Retired • 11 Archived • 12 Accepted • 13 Draft • 14 Build • 15 Validate • 16 ProductionReady • 17 Production • 18 Sunset • 19 PostProduction • 20 Inventory • 21 Development • 22 Offline
majorVersion	Integer		Major version of the product, and generally specified as the first number in a version string (for example, in WebSphere® 6.1, the '6' is the major version).
modifier	Integer		Version specification that is normally tied to fixes within a software release, and is normally specified third in a version string. The Modifier may not always be specified, as in WebSphere 6.1.

Table 438. operatingSystem table (continued)			
Column name	**Type**	**Constraints**	**Description**
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
osConfidence	Integer		Field used by IBM Tivoli Application Dependency Discovery Manager.
osMode	255-character string		Current kernel bit architecture mode of the operating system.
osName	255-character string		String representation of the operating system name.
osVersion	255-character string		Raw text representation of the operating system version.
osId	Integer		Field used by IBM Tivoli Application Dependency Discovery Manager.
osRelease	Integer		Raw text representation of the operating system release.
systemGuid	255-character string		Globally Unique Identifier (GUID) for the operating system.
versionString	255-character string		Complete version specification of the entity, expressed as a single string.
virtualMemorySize	64-bit integer		Allocated size of memory that does not include physical memory size.

ospfArea

The ospfArea table models an OSPF area.

The following table describes the ospfArea table.

Table 439. ospfArea table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of an OSPF area entity from the `entityData` table.
areaId	15-character string	Not null	Identifier for the OSPF area.
isNSSA	8-bit integer		Indicates whether this is an OSPF not-so-stubby area (NSSA).
isExtArea	8-bit integer		Indicates whether this is an OSPF external area.

ospfEndPoint

The ospfEndPoint table represents an OSPF end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

The following table describes the ospfEndPoint table.

Table 440. ospfEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of an OSPF endpoint entity from the `entityData` table.
areaID	15-character string	Not null	
ospfIfAdminState	32-bit integer		
ospfIfState	Enumerated value	Takes one of the following values: 1: down 2: loopback 3: waiting 4: pointToPoint 5: designated-Router 6: backup-DesignatedRouter 7: other-DesignatedRouter	The state of the OSPF interface.
ospfIfType	Enumerated value	Takes one of the following values: 1: broadcast 2: nbma 3: pointTo-Point 5: pointTo-Multipoint	The OSPF interface type.
defaultCost	32-bit integer		

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

ospfNetworkLSA

The ospfNetworkLSA table represents an OSPF Link-State Advertisement (LSA) and includes relevant data.

The following table describes the ospfNetworkLSA table.

Table 441. ospfNetworkLSA table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of an OSPF LSA entity from the `entityData` table.
linkStateId	15-character string	Not null	Specifies link state ID.
networkMask	15-character string		Specifies network mask.
networkType	10-character string		Indicates the network type.

ospfRoutingDomain

The ospfRoutingDomain table represents an OSPF routing domain.

The following table describes the ospfRoutingDomain table.

Table 442. ospfRoutingDomain table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of a network pipe entity from the `entityData` table.
ospfDomain	32-bit integer	Not null	The domain of the OSPF.

ospfService

The ospfService table represents an OSPF service and includes relevant protocol data. This OSPF service runs on a device, as modeled in the hostedService table.

The following table describes the ospfService table.

Table 443. ospfService table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of an OSPF service entity from the `entityData` table.
routerId	15-character string	Not null	The entity ID of the router on which this OSPF service is running.
isAreaBdrRtr	8-bit integer		Indicates whether this router is an area border router.

Table 443. ospfService table (continued)			
Column name	**Type**	**Constraints**	**Description**
isAsBdrRtr	8-bit integer		Indicates whether this router is an AS border router.
isDrRtr	8-bit integer		Indicates whether this router is acting as a designated router.
isBdrRtr	8-bit integer		Indicates whether this router is acting as a backup designated router.
isDrOtherRtr	8-bit integer		Indicates that this router is neither a designated router nor a backup designated router.

Related reference

hostedService

A *hosted service* is a service or application running on a specific main node device. The hostedService table maps a main node device, the *hosting entity*, to the service or applications that are running on that device, the *hosted entities*. The hostedService table belongs to the category *entities*.

pcrfFunction

The `pcrfFunction` table models the Policy and Charging Rules Function (PCRF). The PCRF manages the policy and charging for uplink and downlink service flows and the permitted EPS bearer QoS.

The following table describes the pcrfFunction table.

Table 444. pcrfFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a pcrfFunction entity from the `entityData` table.
pcrfFunctionName	64-character string	NOT NULL	Name of the pcrfFunction (Policy Control and Charging Rules Function) instance configured on the physical node that implements the pcrfFunction
MCC	3-character string		A PCRF can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the PCRF. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the PCRF. The MCC consists of three digits.

Table 444. pcrfFunction table (continued)

Column name	Type	Constraints	Description
MNC	3-character string		A PCRF can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the PCRF. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the PCRF. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		This is a count of the number of PLMNs (Public Land Mobile Networks) supported by the PCRF.
emsDistinguishedName	255-character string		Distinguished name by which the PCRF is known to its element management system (EMS)
emsIpAddress	39-character string		IP address of the element management system
vendorName	64-character string		Vendor or manufacturer of the PCRF
vendorModuleType	64-character string		Vendor specific PCRF Type
softwareVersion	64-character string		Vendor specific PCRF sofware version
operationalState	Enumeration		The operational state of the pcrfFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown

Table 444. pcrfFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
administrativeState	Enumeration		The administrative state of the pcrfFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

pgwFunction

The pgwFunction table models the Packet Data Network Gateway, which provides user plane connectivity to packet data networks. The role of the PGW is implemented within a network hardware node and is modelled by NCIM using the pgwFunction entity type. Multiple pgwFunction instances can be implemented within a single network hardware node.

The following table describes the pgwFunction table.

Table 445. pgwFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a pgwFunction location entity from the entityData table.
pgwFunctionName	64-character string	NOT NULL	Name of the PGW function instance configured on the physical node.
MCC	3-character string		A PGW can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the PGW. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the PGW. The MCC consists of three digits.
MNC	3-character string		A PGW can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the PGW. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the PGW. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		This is a count of the number of PLMNs (Public Land Mobile Networks) supported by the PGW
emsDistinguishedName	255-character string		The distinguished name by which the PGW is known to its element management system (EMS)

Table 445. pgwFunction table (continued)			
Column name	**Type**	**Constraints**	**Description**
emsIpAddress	39-character string		The IP address of the element management system
vendorName	64-character string		The vendor/manufacturer of the PGW
vendorModuleType	64-character string		Vendor specific PGW Type
softwareVersion	64-character string		Vendor specific PGW sofware version
operationalState	Enumeration		Operational state of the pgwFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the pgwFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

physicalBackplane

The `physicalBackplane` table stores the attributes of chassis entities.

The following table describes the `physicalBackplane` table.

Table 446. physicalBackplane table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a chassis entity from the `entityData` table.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
manufacturingDate	Timestamp		Date of manufacture for this entity.

Table 446. physicalBackplane table (continued)			
Column name	**Type**	**Constraints**	**Description**
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
relativePosition	32-bit integer		Indication of the relative position of this entity within the containment.
swRevision	255-character string		An indication of the vendor-specific hardware type of the physical entity.
serialNumber	255-character string		The serial number of the entity.
cdmType	32-bit integer		The model name of the entity.
physicalIndex	32-bit integer		The physical index for this entity.
vendorType	255-character string		Vendor assigned type information.

physicalCard

The `physicalCard` table represents card entities.

The following table describes the `physicalCard` table.

Table 447. physicalCard table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a module (card) entity from the `entityData` table.
aisle	255-character string		Column or longitudinal division of an interior area.

Table 447. physicalCard table (continued)

Column name	Type	Constraints	Description
altitude	FLOAT		Column or longitudinal division of an interior area.
cardConfigurationState	NUMBER(3)		Default value for this field is 0.
fruNumber	255-character string		Specifies the number assigned to a FRU (field-replaceable unit) by the manufacturer.
fruSerialNumber	255-character string		Specifies the serial number assigned to a FRU (field-replaceable unit) by the manufacturer.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
manufacturingDate	Timestamp		Date of manufacture for this entity.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
partNumber	255-character string		Orderable part number for this entity.
rfidTag	255-character string		Radio frequency ID tag identifier.
rackPosition	NUMBER(10)		Particular vertical position on a data center rack.

Table 447. physicalCard table (continued)

Column name	Type	Constraints	Description
relativePosition	32-bit integer		Indication of the relative position of this entity within the containment.
cdmRow	255-character string		Latitudinal division of an interior area.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
systemBoardUUID	255-character string		Specifies the burned-in Globally Unique Identifier (GUID) of this piece of equipment.
cdmType	NUMBER(3)	NOT NULL	Default value for this field is 8.
userTracking	64-character string		Common language location identification (CLLI) code.
xCoordinate	255-character string		Angular distance (east and west) from the prime meridian on the earth's surface.
yCoordinate	255-character string		Angular distance (north and south) from the equator on the earth's surface.
physicalIndex	32-bit integer		The physical index for this entity.
vendorType	255-character string		Vendor assigned type information.
softwareImage	100-character string		
isFRU			Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False

Table 447. physicalCard table (continued)

Column name	Type	Constraints	Description
operStatus	Enumerated value		Operational status of this card. Takes one of the following values: • unknown • ok • disabled • okButDiagFailed • boot • selfTest • failed • missing • mismatchWithParent • mismatchConfig • diagFailed • dormant • outOfServiceAdmin • outOfServiceEnvTemp
adminStatus	Enumerated value		Administrative status of this card. Takes one of the following values: • unknown • enabled • disabled • reset • outOfServiceAdmin
numItemsSupported	32-bit integer		The number of items supported by this entity. For example, if the entity models a layer 1 card, then this number indicates the number of cards supported on the entity. Negative values are ignored.
status	255-character string		Status of this entity.
primaryState	255-character string		Primary state of this entity. This field only applies where the module is the card of a layer 1 device.

Table 447. physicalCard table (continued)			
Column name	**Type**	**Constraints**	**Description**
secondaryState	255-character string		Secondary state of this entity. This field only applies where the module is the card of a layer 1 device.
cardNumber	32-bit integer		Card number.

physicalChassis

The `physicalChassis` table stores the attributes of chassis entities.

The following table describes the `physicalChassis` table.

Table 448. physicalChassis table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a chassis entity from the `entityData` table.
aisle	255-character string		Column or longitudinal division of an interior area.
altitude	FLOAT		Vertical height above sea level at the particular geographical location.
chassisUUID	255-character string		Unique identifier of this chassis.
fruNumber	255-character string		Specifies the number assigned to a FRU (field-replaceable unit) by the manufacturer.
fruSerialNumber	255-character string		Specifies the serial number assigned to a FRU (field-replaceable unit) by the manufacturer.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
manufacturingDate	Timestamp		Date of manufacture for this entity.

Table 448. physicalChassis table (continued)

Column name	Type	Constraints	Description
model	255-character string		The model name of the entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
partNumber	255-character string		Orderable part number for this entity.
rfidTag	255-character string		Radio frequency ID tag identifier.
rackPosition	255-character string		Particular vertical position on a data center rack.
relativePosition	NUMBER(10)		Indication of the relative position of this entity within the containment.
cdmRow	255-character string		Latitudinal division of an interior area.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
systemBoardUUID	255-character string		Specifies the burned-in Globally Unique Identifier (GUID) of this piece of equipment.
cdmType	NUMBER(3)	NOT NULL	Default value for this field is 2.
userTracking	255-character string		Common language location identification (CLLI) code.

Table 448. physicalChassis table (continued)

Column name	Type	Constraints	Description
xCoordinate	255-character string		Angular distance (east and west) from the prime meridian on the earth's surface.
yCoordinate	255-character string		Angular distance (north and south) from the equator on the earth's surface.
physicalIndex	32-bit integer		The physical index for this entity.
vendorType	100-character string		Vendor assigned type information.
className	32-character string	NOT NULL	The name of a class of devices. The master className field is in the entityClass table.
upTime	32-bit integer		The time (in hundredths of a second) since the network management portion of the system was last reinitialized.

Table 448. physicalChassis table (continued)

Column name	Type	Constraints	Description
services	100-character string		A value that indicates the set of services that this entity potentially offers. The value is a sum that initially takes the value zero. Then, for each layer, L, in the range 1 through 7, that this node performs transactions for, 2 raised to (L - 1) is added to the sum. For example, a node that performs only routing functions would have a value of 4 (2^(3-1)). A node that is a host offering application services would have a value of 72 (2^(4-1) + 2^(7-1)). For the Internet suite of protocols, values should be calculated accordingly: • Layer 1: Physical, for example repeaters) • Layer 2: Datalink or subnetwork, for example bridges • Layer 3: Internet, for example supports IP • Layer 4: End-to-end, for example supports TCP • Layer 7: Applications, for example supports the SMTP For systems including OSI protocols, layers 5 and 6 can also be considered.
interfaceCount	32-bit integer		The number of network interfaces (regardless of their current state) present on this system.

Table 448. physicalChassis table (continued)			
Column name	**Type**	**Constraints**	**Description**
isIpForwarding	16-character string		Indication of whether this entity is acting as an IP gateway in respect to the forwarding of datagrams received by this entity but not addressed to this entity. IP gateways forward datagrams, whereas IP hosts do not, unless the source is routed through the host. Takes one of the following values: • forwarding • not-forwarding
accessIPAddress	39-character string		The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.
accessProtocol	Enumerated type (string 7 chars)		String representation of the network protocol. Takes one of the following values: • Unknown • IPv4 • IPv6 • EMSKey
discoveryTime	Timestamp		Time at which the Details agent attempted to discover the device. This value is stored even if the device is not accessible using SNMP.

Related reference

entityClass
The entityClass table stores information on all device classes and relationships between device classes. The table belongs to the category *entities*.

mappings
The `mappings` table provides a means of looking up an alternative textual name. It is used to map non-human-readable data to human-readable data. The `mappings` table belongs to the category *mapping*.

physicalConnector

The `physicalConnector` table stores information about physical connectors.

The following table describes the `physicalConnector` table.

Table 449. physicalConnector table

Column name	Type	Constraints	Description
entityId	32-bit integer	Not null	The identifier of a geographical location from the `entityData` table.
fwRevision	64character string		Firmware version for this entity.
hwRevision	64character string		Hardware version for this entity.
manufacturer	32-bit integer		Vendor-specific hardware type for this entity.
manufacturingDate	Timestamp		Date of manufacture for this entity.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
relativePosition	32-bit integer		Indication of the relative position of this entity within the containment.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
cdmType	Tiny integer	Not null	
physicalIndex	32-bit integer		The physical index for this entity.
vendorType	100-character string		Vendor assigned type information.

physicalFan

The physicalFan table represents fan cooling unit entities.

The following table describes the physicalFan table.

Table 450. physicalFan table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of a fan entity from the entityData table.
fruNumber	255-character string		Specifies the number assigned to a FRU (field-replaceable unit) by the manufacturer.
fruSerialNumber	255-character string		Specifies the serial number assigned to a FRU (field-replaceable unit) by the manufacturer.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
manufacturingDate	timestamp		Date of manufacture for this entity.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
partNumber	255-character string		Orderable part number for this entity.

Table 450. physicalFan table (continued)

Column name	Type	Constraints	Description
relativePosition	NUMBER(10), This format is NUMBER (*precision*,*scale*), where *precision* is the number of digits in a number and *scale* is the number of digits to the right of the decimal point.		Indication of the relative position of this entity within the containment.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
cdmType	NUMBER(3) This format is NUMBER (*precision*,*scale*), where *precision* is the number of digits in a number and *scale* is the number of digits to the right of the decimal point.	Not null	By default, this field takes the value 6
uuid	255-character string		Unique identifier of this piece of equipment.
physicalIndex	NUMBER(10), This format is NUMBER (*precision*,*scale*), where *precision* is the number of digits in a number and *scale* is the number of digits to the right of the decimal point.		The physical index for this entity.
isFRU	Enumerated value		Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False
vendorType	255-character string		Vendor assigned type information.

physicalOther

The `physicalOther` table stores attributes of a component whose physical entity class is known, but does not match any of the supported values.

The following table describes the physicalOther table.

Table 451. physicalOther table

Column name	Type	Constraints	Description
entityId	32-bit integer	Foreign key Not null	The identifier of a network pipe entity from the `entityData` table.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
powerOperStatus	Enumerated value		Operation status. Takes one of the following values: • unknown • offEnvOther • on • offAdmin • offDenied • offEnvPower • offEnvTemp • offEnvFan

Table 451. physicalOther table (continued)			
Column name	**Type**	**Constraints**	**Description**
powerAdminStatus	Enumerated value		Administrative status. Takes one of the following values: • unknown • on • off • inlineAuto • inlineOn
relativePosition	32-bit integer		Indication of the relative position of this entity within the containment.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
cdmType	NUMBER(3)	NOT NULL	By default, takes the value 0.
physicalIndex	NUMBER(10)		The physical index for this entity.
physicalClass	Enumerated value		Takes one of the following values: • 1: unknown • 2: other
vendorType	255-character string		Vendor assigned type information.

physicalPowerSupply

The `physicalPowerSupply` table represents a power supply unit (PSU) entity.

The following table describes the physicalPowerSupply table.

Table 452. physicalPowerSupply table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a network pipe entity from the `entityData` table.
fruNumber	255-character string		Specifies the number assigned to a FRU (field-replaceable unit) by the manufacturer.

Table 452. physicalPowerSupply table (continued)

Column name	Type	Constraints	Description
fruSerialNumber	255-character string		Specifies the serial number assigned to a FRU (field-replaceable unit) by the manufacturer.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
manufacturingDate	Timestamp		Date of manufacture for this entity.
model	255-character string		Model name for this entity.
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
partNumber	255-character string		Orderable part number for this entity.
relativePosition	NUMBER(10),		Indication of the relative position of this entity within the containment.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
cdmType	NUMBER(3)	NOT NULL	By default, this field takes the value of 5.
uuid	255-character string		Unique identifier of this piece of equipment.

Table 452. physicalPowerSupply table (continued)			
Column name	**Type**	**Constraints**	**Description**
physicalIndex	NUMBER(10)		The physical index for this entity.
isFRU	Enumerated value		Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False
powerOperStatus	Enumerated value		See the values for cefcFRUPowerOperStatus in Eumerations table.
powerAdminStatus	Enumerated value		See the values for cefcFRUPowerAdminStatus in Eumerations table.
vendorType	255-character string		Vendor assigned type information.

physicalSensor

The `physicalSensor` table represents sensor entities.

The following table describes the physicalSensor table.

Table 453. physicalSensor table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a sensor entity from the `entityData` table.
aisle	255-character string		Column or longitudinal division of an interior area.
altitude	Floating-point number		Vertical height above sea level at the particular geographical location.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
model	255-character string		Model name for this entity.

Table 453. physicalSensor table (continued)

Column name	Type	Constraints	Description
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
rackPosition	255-character string		Particular vertical position on a data center rack.
relativePosition	NUMBER(10)		Indication of the relative position of this entity within the containment.
cdmRow	255-character string		
swRevision	255-character string		Software revision.
sensorID	255-character string		Identifier for a sensor.
serialNumber	255-character string		The serial number of the entity.
cdmType	NUMBER(3)		The default value for this field is 7.
xCoordinate	255-character string		Angular distance (east and west) from the prime meridian on the earth's surface.
yCoordinate	255-character string		Angular distance (north and south) from the equator on the earth's surface.
physicalIndex	NUMBER(10)		The physical index for this entity.

Table 453. physicalSensor table (continued)

Column name	Type	Constraints	Description
sensorType	Enumerated value		Sensor type. Takes one of the following values: • other • unknown • voltsAC • voltsDC • amperes • watts • hertz • celsius • percentRH • rpm • cmm • truthValue • specialEnum
sensorScale	Enumerated value		Sensor scale. Takes one of the following values: • unknown • yocto • zepto • atto • femto • pico • nano • micro • milli • Units • kilo • mega • giga • tera • exa • peta • zetta • yotta

Table 453. physicalSensor table (continued)			
Column name	**Type**	**Constraints**	**Description**
sensorStatus	Enumerated value		Sensor status. Takes one of the following values: • ok • unavailable • nonoperational • unknown
sensorValue	255-character string		The value for the sensor.
vendorType	255-character string		Vendor assigned type information.

physicalSlot

The `physicalSlot` table represents slot entities.

If you want this table to be populated with MIB data, you must configure the Entity agent to run during the discovery process. The Entity agent discovers detailed containment information from the Entity MIB. By default, the Entity agent is configured not to run during discovery.

The following table describes the physicalSlot table:

Table 454. physicalSlot table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a slot entity from the `entityData` table.
fwRevision	255-character string		Firmware version for this entity.
hwRevision	255-character string		Hardware version for this entity.
manufacturer	255-character string		Vendor-specific hardware type for this entity.
model	255-character string		Model name for this entity.

Table 454. physicalSlot table (continued)

Column name	Type	Constraints	Description
name	255-character string		The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.
slotNumber	NUMBER(10)		Slot number.
relativePosition	NUMBER(10)		Indication of the relative position of this entity within the containment.
swRevision	255-character string		Software revision.
serialNumber	255-character string		The serial number of the entity.
slotState			Current state of the slot.
cdmType	NUMBER(3)		The default value for this field is 7.
powerRedundancy Mode	17-character string		Takes one of the following values: • unknown • notSupported • redundant • combined
physicalIndex	NUMBER(10)		The physical index for this entity.
numItemsSupported	NUMBER		The number of items supported by this slot. For example, if the slot models a layer 1 shelf, then this number indicates the number of cards supported on the shelf. Negative values are ignored.
status	255-character string		Status of this entity.

Table 454. physicalSlot table (continued)			
Column name	**Type**	**Constraints**	**Description**
primaryState	255-character string		Primary state of this entity. This field only applies where the slot is the rack or shelf of a layer 1 device.
secondaryState	255-character string		Secondary state of this entity. This field only applies where the slot is the rack or shelf of a layer 1 device.
vendorType	255-character string		Vendor assigned type information.

pimEndpoint

The pimEndPoint table represents the Protocol Independent Multicast (PIM) end points discovered in the network and their associated attributes.

The following table describes the pimEndPoint table.

Table 455. pimEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a PIM end point from the `entityData` table.
pimmode	Enumerated value	Takes one of the following values: • unknown • sparse • dense • sparseDense	The PIM mode.
designatedRouter	15-character string		IP address of the Designated Router.
helloInterval	32-bit integer		Frequency (seconds) of PIM Hello message transmission.
joinPruneInterval	32-bit integer		Frequency (seconds) of PIM join/prune messages
csbrPreference	32-bit integer		Candidate BSR preference value. A value of -1 means it is not a BSR candidate.
isCandidateRP	8-bit integer		Indicates that the end point acts as a Candidate RP.
rpCandidateGroup	15-character string		IP address of multicast group for which this end point is a Candidate RP.

Table 455. pimEndPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
rpCandidateMask	15-character string		Mask of multicast group for which this end point is a Candidate RP
crpHoldTime	32-bit integer		The hold time for the Candidate RP. A value of 0 indicates that this end point is not an RP candidate.
bsrAddress	15-character string		Bootstrap Router IP address
bsrExpiryTime	32-bit Integer		Time remaining until BSR is considered down (and a new one selected).

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

pimNetwork

The pimNetwork table holds the Protocol Independent Multicast (PIM) Network collection entity which collects all PIM-enabled routers.

The following table describes the pimNetwork table.

Table 456. pimNetwork table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of the PIM Network collection entity.

pimService

The pimService table represents a Protocol Independent Multicast (PIM) service and includes relevant protocol data.

The following table describes the pimService table.

Table 457. pimService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a PIM service entity from the `entityData` table.
joinPruneInterval	32-bit integer		The PIM join/prune message interval (in seconds).

Table 457. pimService table (continued)			
Column name	**Type**	**Constraints**	**Description**
isRP	8-bit integer		Indicates whether the service is known to be acting as a Rendezvous Point for one or more Multicast Groups.
isCRP	8-bit integer		Indicates whether the service acts as a Candidate Rendezvous Point for one or more Multicast Groups.
isBSR	8-bit integer		Indicates whether the service is known to be acting as a Bootstrap Router.
isCBSR	8-bit integer		Indicates whether the service acts as a Candidate Bootstrap Router.
isDR	8-bit integer		Indicates whether the service acts as a Designated Router.

plmn

The `plmn` table models a Public Land Mobile Network (PLMN). A PLMN is a network that provides land mobile telecommunications services to the public. Each operator providing mobile services has its own PLMN.

The following table describes theplmn table.

Table 458. plmn table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a plmn entity from the `entityData` table.
plmnName	64-character string		Name or description of the PLMN
MCC	3-character string		Specifies the Mobile Country Code (MCC) of the PLMN. The MCC consists of three digits.
MNC	3-character string		Specifies the Mobile Network Code (MNC) of the PLMN. The length of the MNC (two or three digits) depends upon the value of the MCC.

portEndPoint

The portEndPoint holds data about TCP/UDP endpoints found by the NMAPScan agent.

The following table describes the portEndPoint table.

Table 459. portEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	Not null Primary key	The entityId of the portEndPoint entity.
portId	Integer	Not null	The numeric port ID.
portState	15-character string		The port state, such as open/closed.
protocol	5-character string		The protocol, whether TCP or UDP.
serviceProduct	255-character string		The name of the port service.
serviceVersion	75-character string		The version if available, such as 5.0.54a-enterprise.
serviceName	75-character string		The service name, typically short, such as ftp.

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

Fix Pack 3 probe

The `probe` table holds probe specific details of all discovered network probes.

The following table describes the `probe` table.

Table 460. probe table			
Column name	**Type**	**Constraints**	**Description**
`adminStatus`	Enumerated value		The administrative status of the probe. Takes one of the following values: • active • inactive • other
`callDuration`	32-bit integer		Duration for RTP/Video probes.
`codecType`	Enumerated value		Codec type used by Jitter probes. Takes one of the following values: • unknown • g711Alaw • g711Ulaw • g729A

Table 460. probe table (continued)

Column name	Type	Constraints	Description
differentiatedService	32-bit integer		Type of service octet value (if set in IP header).
entityId	32-bit integer	Primary Key Foreign Key (entityData table) Not Null	Unique identifier for the entity.
frequency	32-bit integer		Duration between probe operations, in seconds.
httpVersion	10-character string		HTTP server version. Used with HTTP probes.
icpifAdvantage	32-bit integer		Used in Jitter probe ICPIF calculations.
name	32-character string		Name of the probe.
nativeType	32-character string		A human-readable textual representation of the nativeTypeId.
nativeTypeId	32-bit integer		An integer representation of the probe type. The values depend on the data definition in the probe.
operStatus	Enumerated value		The operational status of the probe. Takes one of the following values: • active • inactive • other
owner	255-character string		Owner/creator of the probe instance.
packetInterval	32-bit integer		The delay between packets, in ms.
probeCount	32-bit integer		Number of packets to transmit.
probeId	255-character string		Unique probe identifier. Identifies the probe uniquely at a device level.
target	255-character string		Destination address. Can be an IP address.

Table 460. probe table (continued)			
Column name	**Type**	**Constraints**	**Description**
targetPort	32-bit integer		Port number on the target.
timeout	32-bit integer		The maximum time to wait for a proper operation to complete, in ms.
source	255-character string		Source address. Can be an IP address.
sourceInterface	32-bit integer		Source interface index.
sourcePort	32-bit integer		The port number on the source.
sourceVoicePort	255-character string		Specifies the voice port on the gateway.
vrfName	255-character string		VRF associated with this probe.

Fix Pack 3 probeCollection

The probeCollection table represents entities that collect probes, or further probe collections. This table is used to provide a hierarchical probe collection and to allow the display of probes and the devices they relate to.

The following table describes the probeCollection table.

Table 461. probeCollection table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary Key Foreign Key (entityData table) Not Null	The entity identifier.

Fix Pack 3 probeEndPoint

The probeEndPoint table provides probe-specific protocol end point information. These end points are implemented by interfaces that are identified as a probe source or target.

The following table describes the probeEndPoint table.

Table 462. probeEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary Key Foreign Key (entityData table) Not Null	Unique identifier for the entity.

Table 462. probeEndPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
role	Enumerated value		The role of this particular endpoint in the related probe configuration. Takes one of the following values: • other • source • target

Fix Pack 3 probeService

The probeService table holds details of the technology that provides network probes for a given device.

The following table describes the probeService table.

Table 463. probeService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary Key Foreign Key (entityData table) Not Null	Unique identifier for the entity.
maxProbes	32-bit integer		Total number of probes supported.
monitorType	16-character string		Name of the technology that provides the probes. For example, IPSLA. This value is used to group probes.
responderEnabled	Enumerated value		Indicates whether the device is configured as a responder. Takes one of the following values: • true • false
updateTime	Timestamp		Time of last update.
version	255-character string		Version of the monitor type.

ranBaseStation

The ranBaseStation table describes Radio Access Network (RAN) base stations.

The following table describes the ranBaseStation table.

Table 464. ranBaseStation table			
Column name	**Type**	**Constraints**	**Description**
baseStationId	varchar(64)	not null	A string identifying the base station. In a naming convention that follows the GSM 04.08 standard, the string consists of the Network Color Code (NCC) and the Base Station Code (BSC).
entityId	Integer	not null	The identifier of a base station entity from the `entityData` table.
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS

ranBaseStationController

The ranBaseStationController table describes Radio Access Network (RAN) base station controllers.

The following table describes the ranBaseStationController table.

Table 465. ranBaseStationController table			
Column name	**Type**	**Constraints**	**Description**
baseStationControllerId	varchar(64)		A string identifying the base station controller. In a naming convention that follows the GSM 04.08 standard, the string consists of the Base Station Color Code (BSC).
entityId	Integer	not null	The identifier of a base station controllers entity from the `entityData` table.
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS

ranCircuitSwitchedCore

The `ranCircuitSwitchedCore` table describes RAN circuit switched core entities, which are collection entities that collect the entities involved in the circuit switched core network of a given mobile phone network.

The following table describes the `ranCircuitSwitchedCore` table.

Table 466. ranCircuitSwitchedCore table

Column name	Type	Constraints	Description
entityId	Integer	not null	The identifier of a RAN circuit switched core entity from the `entityData` table.
mcc	varchar(64)		Mobile country code, which consists of three digits. The MCC uniquely identifies the country of domicile of the mobile subscriber.
mnc	varchar(64)		Mobile network code, which consists of two or three digits for GSM and UMTS applications. The MNC identifies the home PLMN (Public Land Mobile Network) of the mobile subscriber. The length of the MNC (two or three digits) depends on the value of the MCC.

ranGGSN

The ranGGSN table describes Radio Access Network (RAN) Gateway GPRS Serving Nodes (GGSNs).

The following table describes the ranGGSN table.

Table 467. ranGGSN table

Column name	Type	Constraints	Description
accessPointName	varchar(64)		The Access Point Name is a unique name that is associated with the IP address of a specific GGSN through a DNS lookup.
entityId	Integer	not null	The identifier of a GGSN entity from the `entityData` table.
ggsnId	varchar(64)		A unique identifier for the GGSN.

Table 467. ranGGSN table (continued)			
Column name	**Type**	**Constraints**	**Description**
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS

ranGSMCell

The ranGSMCell table describes Radio Access Network (RAN) GSM cells.

The following table describes the ranGSMCell table.

Table 468. ranGSMCell table			
Column name	**Type**	**Constraints**	**Description**
bcc	integer		The Base station Color Code (BCC), which is used to distinguish neighboring cells of the same operator on the same channel.
broadcastPower	varchar(64)		The broadcast channel power level (dBm).
broadcastScrambling Code	integer	From 0 to 500.	Used to generate the primary scrambling code.
cellId	varchar(64)		Unique identifier for the cell.
entityId	Integer	not null	The identifier of a GSM cell entity from the `entityData` table.
hopSeqNum	integer	From 0 to 63.	The hopping sequence number. Defines the set of channels that the cell is to use for frequency hopping.
msTxPower	integer		The maximum power level that the mobile station is allowed to use.
ncc	integer		The Network Color Code (NCC) is used to distinguish neighboring cells between operators of different countries broadcasting channel.
racc	integer		The Routing Area Color Code.

Table 468. ranGSMCell table (continued)			
Column name	**Type**	**Constraints**	**Description**
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS
rxLevAccessMin	integer		The minimum Rx signal strength threshold.
tsc	integer	From 0 to 7.	Specifies the Training Sequence Code (TSC) used in the cell.

ranLocationArea

The ranLocationArea table describes Radio Access Network (RAN) location areas, in which there may be one or more GSM/UMTS cells. This entity is a collection and it models those devices within which a given mobile user can move for voice access before having to switch to a different location area.

The following table describes the ranLocationArea table.

Table 469. ranLocationArea table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a RAN location area entity from the `entityData` table.
lac	varchar(64)	not null	The Location Area Code (LAC) identifies a location area within a PLMN.
mcc	varchar(64)	not null	The three-digit Mobile Country Code (MCC) uniquely identifies the country of the mobile subscriber.
mnc	varchar(64)	not null	The Mobile Network Code (MNC) identifies the home PLMN (Public Land Mobile Network) of the mobile subscriber. The length of the MNC (two or three digits) depends on the value of the MCC (Mobile Country Code).

ranMediaGateway

The ranMediaGateway table describes Radio Access Network (RAN) media gateways.

The following table describes the ranMediaGateway table.

Table 470. ranMediaGateway table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a media gateway entity from the `entityData` table.
mgwId	varchar(64)		Unique identifier for the media gateway.

ranMobileSwitchingCentre

The ranMobileSwitchingCentre table describes Radio Access Network (RAN) Mobile Switching Centers.

The following table describes the ranMobileSwitchingCentre table.

Table 471. ranMobileSwitchingCentre table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a transceiver entity from the `entityData` table.
mscId	varchar(64)		A unique, enterprise-specific identifier for the mobile switching center.
mscType	varchar(10)		The type of the mobile switching centers. Possible values are: • Unknown • Other • Voice Switch • MSCS • Type2G3G
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS

ranMSS

The ranMSS table describes Radio Access Network (RAN) mobile switching center servers.

The following table describes the ranMSS table.

Table 472. ranMSS table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a mobile switching center server entity from the `entityData` table.
mssId	varchar(64)		Unique identifier for the MMS.

ranNodeB

The `ranNodeB` table describes Node B entities.

The following table describes the `ranNodeB` table.

Table 473. ranNodeB table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a Node B entity from the `entityData` table.
ranTechnologyType	varchar(10)		The wireless technology type for this entity, which can take any of the following values: • 0: Unknown • 1: Other • 2: GSM • 3: GPRS • 4: UMTS
nodeBId	varchar(64)	not null	Node B entity identifier string

ranNodeBLocalCell

The `ranNodeBLocalCell` table models the local Node B identifier. This identifier is related to local hardware that is available to manage a given cell.

The following table describes the `ranNodeBLocalCell` table.

Table 474. ranNodeBLocalCell table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a Node B local cell entity from the `entityData` table.
ranTechnologyType	varchar(10)		The wireless technology type for this entity, which can take any of the following values: • 0: Unknown • 1: Other • 2: GSM • 3: GPRS • 4: UMTS

Table 474. ranNodeBLocalCell table (continued)			
Column name	**Type**	**Constraints**	**Description**
localCellId	varchar(64)	not null	Node B local cell entity identifier string

ranPacketControlUnit

The ranPacketControlUnit table describes Radio Access Network (RAN) base station controllers.

The following table describes the ranPacketControlUnit table.

Table 475. ranPacketControlUnit table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a transceiver entity from the `entityData` table.
pcuId	varchar(64)		Unique internal identifier for the Packet Control Unit. In some networks, this is the same as the Base Station Controller ID.
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS

ranPacketSwitchedCore

The `ranPacketSwitchedCore` table describes RAN packet switched core entities, which are collection entities that collect the entities involved in the packet switch core network of a given mobile phone network.

The following table describes the `ranPacketSwitchedCore` table.

Table 476. ranPacketSwitchedCore table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a RAN packet switched core entity from the `entityData` table.
mcc	varchar(64)		Mobile country code, which consists of three digits. The MCC uniquely identifies the country of domicile of the mobile subscriber.

Table 476. ranPacketSwitchedCore table (continued)			
Column name	**Type**	**Constraints**	**Description**
mnc	varchar(64)		Mobile network code, which consists of two or three digits for GSM and UMTS applications. The MNC identifies the home PLMN (Public Land Mobile Network) of the mobile subscriber. The length of the MNC (two or three digits) depends on the value of the MCC.

ranRadioCore

The `ranRadioCore` table describes RAN radio core entities, which are collection entities that collect the entities involved in the RAN network radio core; that is, radio network controller (RNC) and base station controller (BSC) entities.

The following table describes the `ranRadioCore` table.

Table 477. ranRadioCore table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a RAN radio core entity from the `entityData` table.
mcc	varchar(64)		Mobile country code, which consists of three digits. The MCC uniquely identifies the country of domicile of the mobile subscriber.
mnc	varchar(64)		Mobile network code, which consists of two or three digits for GSM and UMTS applications. The MNC identifies the home PLMN (Public Land Mobile Network) of the mobile subscriber. The length of the MNC (two or three digits) depends on the value of the MCC.

ranRadioNetworkController

The `ranRadioNetworkController` table describes RAN radio network controller entities.

The following table describes the `ranRadioNetworkController` table.

Table 478. ranRadioNetworkController table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a RAN radio network controller entity from the `entityData` table.
ranTechnologyType	varchar(10)		The wireless technology type for this entity, which can take any of the following values: • 0: Unknown • 1: Other • 2: GSM • 3: GPRS • 4: UMTS
rncId	varchar(64)	not null	RAN radio network controller entity identifier string. A unique identifier in the network in which the RNC is operational.

ranRoutingArea

The `ranRoutingArea` table describes RAN routing area entities, which are devices within which a given mobile user can move for data access before having to switch to a different routing area.

The following table describes the `ranRoutingArea` table.

Table 479. ranRoutingArea table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a RAN routing area entity from the `entityData` table.
rac	varchar(10)	not null	Enterprise-specific routing area code.
mcc	varchar(64)	not null	Mobile country code, which consists of three digits. The MCC uniquely identifies the country of domicile of the mobile subscriber.
mnc	varchar(64)	not null	Mobile network code, which consists of two or three digits for GSM and UMTS applications. The MNC identifies the home PLMN (Public Land Mobile Network) of the mobile subscriber. The length of the MNC (two or three digits) depends on the value of the MCC.

Table 479. ranRoutingArea table (continued)			
Column name	**Type**	**Constraints**	**Description**
lac	varchar(64)	not null	Location area code, which is a fixed length code (of 2 octets) identifying a location area within a PLMN.

ranSector

TheranSector table describes Radio Access Network (RAN) cell sectors.

The following table describes the ranSector table.

Table 480. ranSector table			
Column name	**Type**	**Constraints**	**Description**
beamDirection	integer		The beam direction of the sector. Degrees 0 – North, 90 East, 180 South, 270 West.
entityId	Integer	not null	The identifier of a cell sector entity from the `entityData` table.
sectorHeight	integer		Height of the sector above ground in centimeters.
sectorId	varchar (64)	not null	Enterprise-specific sector identifier.

ranSGSN

The ranSGSN table describes Radio Access Network (RAN) Serving GPRS Serving Nodes (SGSNs).

The following table describes the ranSGSN table.

Table 481. ranSGSN table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	not null	The identifier of a SGSN entity from the `entityData` table.
ranTechnologyType	varchar(10)		The type of wireless technology used by the base station. Possible values are: • Unknown • Other • GSM • GPRS • UMTS
sgsnId	varchar(64)	not null	Unique identifier for the SGSN.

ranTransceiver

The ranTransceiver table describes transceivers.

The following table describes the ranTransceiver table.

Table 482. ranTransceiver table

Column name	Type	Constraints	Description
entityId	Integer	not null	The identifier of a transceiver entity from the `entityData` table.
transceiverType	varchar(10)		The type of transceiver, which can be any of the following values: • 0: Unknown • 1: Other • 2: Normal • 3: Dedicated • 4: Extended
trxId	varchar(64)	not null	Transceiver identifier string

ranUtranCell

The ranUtranCell table describes Radio Access Network (RAN) UTRAN cells.

The following table describes the ranUtranCell table.

Table 483. ranUtranCell table

Column name	Type	Constraints	Description
broadcastPower	varchar(64)		The broadcast channel power level (dBm).
broadcastScrambling Code	integer	From 0 to 500.	Used to generate a number for code scrambling.
cellId	varchar(64)	not null	Unique identifier for the cell.
entityId	Integer	not null	The identifier of a transceiver entity from the `entityData` table.
maxTransmission Power	integer		Maximum transmission power for all downlink channels that are allowed to be used simultaneously in a cell, added together Unit: 0.1 dBm Range 0-500.
primarySchPower	integer		Primary synchronization power. Unit: 0.1 dB Range: 350-150.
primaryScrambling Code	long		A code used to separate the transmission of one cell from another.

Table 483. ranUtranCell table (continued)			
Column name	**Type**	**Constraints**	**Description**
secondarySchPower	integer		Secondary synchronization power. Unit: 0.1 dB Range: 350-150.
uarfcnDL	integer		Absolute downlink radio frequency number.
uarfcnUL	integer		Absolute uplink radio frequency number.

rtExportList

The rtExportList table stores export route targets associated with Virtual Forwarding and Routing (VRF).

The following table describes the rtExportList table.

Table 484. rtExportList table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a route target export list entity from the `entityData` table.
routeTarget	64-character string	Not null	Router target value.

rtImportList

The rtImportList table stores import route targets associated with Virtual Forwarding and Routing (VRF).

The following table describes the rtImportList table.

Table 485. rtImportList table			
Column Name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a route target import list entity from the `entityData` table.
routeTarget	64-character string	Not null	The router target value.

sgwFunction

The `sgwFunction` table models the Serving Gateway (SGW), which resides in the user plane where it forwards and routes packets to and from the eNodeB and packet data network gateway (PGW). The role of the SGW is implemented within a network hardware node and is modelled by NCIM using the sgwFunction entity type. Multiple sgwFunction instances can be implemented within a single network hardware node.

The following table describes the `sgwFunction` table.

Table 486. sgwFunction table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of an sgwFunction location entity from the `entityData` table.

Table 486. sgwFunction table (continued)

Column name	Type	Constraints	Description
sgwFunctionName	64-character string	NOT NULL	Name of the SGWFunction instance configured on Physical node that implement the SGWFunction
MCC	3-character string		An SGW can support multiple PLMNs. The MCC attribute specifies the Mobile Country Code (MCC) of the primary PLMN supported by the SGW. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the SGW. The MCC consists of three digits.
MNC	3-character string		An SGW can support multiple PLMNs. The MNC attribute specifies the Mobile Network Code (MNC) of the primary PLMN supported by the SGW. Primary PLMN usually means the sole PLMN or the PLMN of the operator responsible for the operation and maintenance of the SGW. The length of the MNC (two or three digits) depends upon the value of the MCC.
supportedPLMNs	Integer		This is a count of the number of PLMNs (Public Land Mobile Networks) supported by the SGW
emsDistinguishedName	255-character string		The distinguished name by which the SGW is known to its element management system (EMS)
emsIpAddress	39-character string		The IP address of the element management system
vendorName	64-character string		The vendor/manufacturer of the SGW
vendorModuleType	64-character string		Vendor specific SGW Type
softwareVersion	64-character string		Vendor specific SGW sofware version
operationalState	Enumeration		Operational state of the sgwFunction. Takes one of the following values: • Enabled • Disabled • Other • Unknown
administrativeState	Enumeration		Administrative state of the sgwFunction. Takes one of the following values: • Unlocked • Locked • Shutting Down • Other • Unknown

snmpSystem

The snmpSystem table represents a Simple Network Management Protocol (SNMP) managed host on a network.

The following table describes the snmpSystem table.

Table 487. snmpSystem table

Column name	Type	Constraints	Description
entityId	Integer	not null	The identifier of an snmpSystem entity from the entityData table.
sysContact	255-character string		The textual identification of the contact person for this managed node, and information on how to contact this person. If no contact information is known, the value is the zero-length string.
sysDescr	255-character string		A textual description of the entity. This value must include the full name and version identification of the system hardware type, software operating-system, and networking software.
sysLocation	255-character string		The physical location of this node, for example "telephone closet, 3rd floor." If the location is unknown, the value is the zero-length string.
sysName	255-character string		An administratively-assigned name for this managed node. By convention, this is the fully-qualified domain name of the node. If the name is unknown, the value is the zero-length string.
sysObjectId	100-character string		The vendor's authoritative identification of the network management subsystem contained in the entity.

subnet

The subnet table represents a logical collection of IP addresses collected within a subnet.

The following table describes the subnet table

Table 488. subnet table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Not null Foreign key	The identifier of a subnet entity from the `entityData` table.
network	39-character string	Not null	The IP address of this subnet.
netmask	39-character string	Not null	The netmask for this subnet.
protocol	String value		An integer representation of the IP protocol used by the presently-defined zone: • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6
netmaskBits	32–bit integer		Netmask bits for the subnet
addressSpace	255–character string		Relevant NAT address space if network address translation is being used.

trackingArea

The `trackingArea` table models an LTE tracking area.

The following table describes the trackingArea table.

Table 489. trackingArea table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a trackingArea location entity from the `entityData` table.
TAC	64-character string		Tracking Area Code (TAC), consisting of 16 bits. This is an identifier for the tracking area and is unique within a public land mobile network (PLMN).
TAI	64-character string	NOT NULL	Tracking Area Identifier (TAI). This is a globally unique tracking area identifier, made up of the PLMN ID and the TAC.
trackingAreaName	64-character string		Name of the tracking area.

transmissionTp

The `transmissionTp` table provides information about transmission interface entities in the network.

The following table describes the `transmissionTp` table.

Table 490. transmissionTp table			
Column name	**Type**	**Constraints**	**Description**
entityId	32–bit integer	Not null	The identifier of a transmission interface entity from the `entityData` table.
tpType	3-character string		The type of transmission interface entity: • Physical termination point (PTP) • Connection termination point (CTP)
primaryState	255-character string		Primary state of this entity.
secondaryState	255-character string		Secondary state of this entity.
layerRate	255-character string		Layer rate of the transmission interface entity.
isEdgePoint	5-character string		Label of the transmission interface entity. Takes one oft he following values: • True • False
mappingMode	255-character string		Mapping mode of the transmission interface entity.

userPlaneViewCollection

The `userPlaneViewCollection` table supports the dynamic collection views under **LTE Network Topology** > **User Plane by Tracking Area** in the Network Views. Each instance of this entity type collects the eNodeBs in the corresponding tracking area, together with the devices that these eNodeBs are connected to on the user plane.

The following table describes the userPlaneViewCollection table.

Table 491. userPlaneViewCollection table			
Column name	**Type**	**Constraints**	**Description**
entityId	Integer	FOREIGN KEY NOT NULL	The identifier of a userPlaneViewCollection entity from the `entityData` table.
viewType	64-character string	NOT NULL	Specifies the type of view.

vlanTrunkEndPoint

The vlanTrunkEndPoint table represents a VLAN trunk end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

The following table describes the vlanTrunkEndPoint table.

Table 492. vlanTrunkEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a VLAN trunk endpoint entity from the `entityData` table.
vlanClass	Enumerated		The class of the VLAN. Possible values are: `cvlan` (Customer VLAN using QinQ), `svlan` (Service VLAN using QinQ), or `local` (VLAN not using QinQ).
vlanId	32-bit integer		The identifier for the VLAN carried by this protocol end point object. If multiple VLANs are carried by the trunk then a vlanTrunkEndPoint entity should be created for each one.
vlanTag	32-bit integer		The tag used for this VLAN. In Cisco devices, this tag is usually the same as the `vlanId` value. However, for other manufacturers, the tag might be different.

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

vpnRouteForwarding

The vpnRouteForwarding table models a VPN routing and forwarding table.

The following table describes the vpnRouteForwarding table.

Table 493. vpnRouteForwarding table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a VPN Routing and Forwarding table entity from the `entityData` table.
VRFName	255-character string		The name of the VPN Routing and Forwarding table.
routeDistinguisher	255-character string	Not null	The route distinguisher for the VPN Routing and Forwarding table.

vpwsEndPoint

The vpwsEndPoint table represents a VPWS end point and includes relevant data. This endpoint is implemented by a physical interface, as modeled in the protocolEndPoint table.

The following table describes the vpwsEndPoint table.

Table 494. vpwsEndPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Primary key Not null	The identifier of a network pipe entity from the `entityData` table.
VCID	64-character string	Primary key Not null	The Virtual Circuit Identifier (VCID) for this entity.
circuitId	128-character string		The ID for this circuit.
circuitType	32-bit integer		The type of circuit.
circuitStatus	32-bit integer		The status of circuit.
inboundLabel	32-bit integer		The inbound label related to this endpoint.
outboundLabel	32-bit integer		The outbound label related to this endpoint.

Related reference

protocolEndPoint

The protocolEndPoint table allows a higher-level connection to be defined in terms of lower-level connections. It associates a device entity, usually an interface, with protocol-specific information associated with that device entity. The protocolEndPoint table belongs to the category *connectivity*.

vtpDomain

The vtpDomain table represents a VLAN trunking protocol domain.

The following table describes the vtpDomain table.

Table 495. vtpDomain table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Foreign key Not null	The identifier of a VTP domain entity from the `entityData` table.
vtpDomainName	64-character string	Not null	The name of this VTP domain.
vtpDomainLocalMode	15-character string		The local mode of this VTP domain.

wlan

The wlan table stores information about wireless networks.

The following table describes the wlan table.

Table 496. wlan table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each WLAN across all domains.
ssid	50-character string	Not null	The SSID of this WLAN.
broadcastSSID	15-character string	Not null	A string indicating whether the broadcast of the SSID of this WLAN is enabled or disabled.

wlanAccessPoint

The wlanAccessPoint table stores information about wireless access points.

The following table describes the wlanAccessPoint table.

Table 497. wlanAccessPoint table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each wireless Access Point across all domains.
apMACAddress	40-character string		The MAC address of this access point.
adminStatus	15-character string		The administrative status of this access point.
apType	25-character string		The type of this access point.
ethernetMAC	40-character string		The Ethernet MAC address of this access point.
gatewayRouter	39-character string		The IP address of the gateway router of this access point.
isStaticIP	15-character string		Whether the IP address is static or not.
lastRebootReason	40-character string		The reason why this access point was last rebooted.
monitorOnlyMode	25-character string		The monitor only mode of this access point.
numEthernetIfaces	32-bit integer		The number of Ethernet interfaces of this access point.
numRadioInterfaces	32-bit integer		The number of radio interfaces of this access point
operationalStatus	25-character string		The operational status of this access point.
snmpIndex	50-character string		The SNMP index of this access point.

Table 497. wlanAccessPoint table (continued)			
Column name	**Type**	**Constraints**	**Description**
statsTimer	32-bit integer		The interval in seconds between each time the access point sends its DOT11 statistics to the WLAN controller.
trafficViaPortNum	32-bit integer		The port number where traffic flows in and out of this access point.
upTime	50-character string		How long the access point have been running since the last power up or reboot.

wlanChannel

The wlanChannel table stores information about wireless channels.

The following table describes the wlanChannel table.

Table 498. wlanChannel table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each wireless channel across all domains.
channelNo	10-character string		The number of this WLAN channel.

wlanDot11Interface

The wlanDot11Interface table stores information about DOT11 interfaces.

The following table describes the wlanDot11Interface table.

Table 499. wlanDot11Interface table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each WLAN DOT11 across all domains.
antennaDiversity	50-character string		The antenna diversity value of this DOT11 interface.
antennaGain	32-bit integer		The antenna gain of this DOT11 interface.
antennaOptions	30-character string		The antenna options of this DOT11 interface.
channelAssignMode	30-character string		The antenna channel assignment mode of this DOT11 interface.

Table 499. wlanDot11Interface table (continued)			
Column name	**Type**	**Constraints**	**Description**
channelNo	10-character string		The channel number of this DOT11 interface.
dot11IfType	20-character string		The interface type of this DOT11 interface.
packetSniffMode	20-character string		The packet sniff mode of this DOT11 interface.
packetSniffChannelNo	10-character string		The packet sniff channel number of this DOT11 interface.
radioSlot	32-bit integer		The radio slot of this DOT11 interface.
snmpIndex	50-character string		The SNMP index of this DOT11 interface.
supportedChannelNo	255-character string		The supported channel number of this DOT11 interface.
trafficViaPortNum	32-bit integer		The port number where traffic flows in and out of this DOT11 interface.
txPowerControlMode	30-character string		The Tx power control mode of this DOT11 interface.
txPowerLevel	32-bit integer		The Tx power level of this DOT11 interface.

wlanService

The wlanService table stores information about wireless LAN services.

The following table describes the wlanService table.

Table 500. wlanService table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each WLAN service across all domains.
aclName	75-character string		The ACL name of this WLAN service.
adminStatus	15-character string		The administrative status of this WLAN service.
authType	15-character string		The authentication type of this WLAN service.
broadcastSSID	15-character string		The broadcast SSID of this WLAN service.
dhcpRequired	15-character string		Whether DHCP server is required or not.

Table 500. wlanService table (continued)			
Column name	**Type**	**Constraints**	**Description**
dhcpServerIP	39-character string		The IP address of the DHCP server.
interfaceName	20-character string		The interface name of this WLAN service.
qos	20-character string		The QoS of this WLAN service.
radioPolicy	15-character string		The radio policy of this WLAN service.
sessionTimeout	32-bit integer		The session timeout in seconds of this WLAN service.
snmpIndex	50-character string		The SNMP index of this WLAN service.
ssid	50-character string	Not null	The SSID of this WLAN service.
wepSecurity	15-character string		Whether the WEP security is enabled or disabled.
wepType	15-character string		The WEP type of this WLAN service.

wlanSpec

The wlanSpec table stores information about wireless specifications.

The following table describes the wlanSpec table.

Table 501. wlanSpec table			
Column name	**Type**	**Constraints**	**Description**
entityId	32-bit integer	Primary key Foreign key Not null	Automatically incremented ID that provides a unique value for each WLAN specification across all domains.
protocol	50-character string	Not null	The protocol of this WLAN specification.

Entity attribute views

Entity attribute views define attributes for the entities discovered by Network Manager, in legacy NCIM topology database format. This enables backward compatibility; for example, SQL queries written to use the legacy NCIM database tables still function because they retrieve data from the entity attribute views.

backplane

The `backplane` view joins data from a number of tables and is equivalent to the `backplane` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `backplane` view.

Table 502. backplane view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalBackplane entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalBackplane Model
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalBackplane RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalBackplane PhysicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalBackplane Name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData Description

chassis

The `chassis` view joins data a number of tables, and is equivalent to the `chassis` table that existed in Network Manager versions 3.9 and earlier, thereby ensuring backward compatibility.

The following table describes the `chassis` view.

Table 503. chassis view

Column name	Description	Containing table and field name
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalChassis entityId
className	The name of a class of devices. The master className field is in the entityClass table.	physicalChassis className
sysName	An administratively-assigned name for this managed node. By convention, this is the fully-qualified domain name of the node. If the name is unknown, the value is the zero-length string.	physicalChassis sysName
sysDescr	A textual description of the entity. This value must include the full name and version identification of the system hardware type, software operating-system, and networking software.	entityData Description
sysObjectId	The vendor's authoritative identification of the network management subsystem contained in the entity.	physicalChassis sysObjectId
sysLocation	The physical location of this node, for example "telephone closet, 3rd floor." If the location is unknown, the value is the zero-length string.	physicalChassis Location
sysContact	The textual identification of the contact person for this managed node, and information on how to contact this person. If no contact information is known, the value is the zero-length string.	physicalChassis Contact
sysUpTime	The time (in hundredths of a second) since the network management portion of the system was last reinitialized.	physicalChassis UpTime

Table 503. chassis view (continued)

Column name	Description	Containing table and field name
sysServices	A value that indicates the set of services that this entity potentially offers. The value is a sum that initially takes the value zero. Then, for each layer, L, in the range 1 through 7, that this node performs transactions for, 2 raised to (L - 1) is added to the sum. For example, a node that performs only routing functions would have a value of 4 (2^(3-1)). A node that is a host offering application services would have a value of 72 (2^(4-1) + 2^(7-1)). For the Internet suite of protocols, values should be calculated accordingly: • Layer 1: Physical, for example repeaters) • Layer 2: Datalink or subnetwork, for example bridges • Layer 3: Internet, for example supports IP • Layer 4: End-to-end, for example supports TCP • Layer 7: Applications, for example supports the SMTP For systems including OSI protocols, layers 5 and 6 can also be considered.	physicalChassis Services
ifNumber	The number of network interfaces (regardless of their current state) present on this system.	physicalChassis InterfaceCount
ipForwarding	Indication of whether this entity is acting as an IP gateway in respect to the forwarding of datagrams received by this entity but not addressed to this entity. IP gateways forward datagrams, whereas IP hosts do not, unless the source is routed through the host. Takes one of the following values: • forwarding • not-forwarding	physicalChassis isIpForwarding
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalChassis Model

Table 503. chassis view (continued)

Column name	Description	Containing table and field name
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalChassis RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalChassis physicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalChassis Name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData Description
serialNumber	The serial number of the entity.	physicalChassis SerialNumber
modelName	The model name of the entity.	physicalChassis Model
orderablePartNumber	Orderable part number for this entity.	physicalChassis PartNumber
hardwareVersion	Hardware version for this entity.	physicalChassis HWRevision
OSType	The operating system type related to this chassis.	operatingSystem OSName

Table 503. chassis view (continued)		
Column name	**Description**	**Containing table and field name**
OSVersion	The operating system version related to this chassis.	operatingSystem OSVersion
OSImage	The operating system image related to this chassis.	operatingSystem VersionString
accessIPAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	physicalChassis accessIP
accessProtocol	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device	physicalChassis accessProtocol
memorySize	The size, in MB, of the available memory.	computerSystem MemorySize
discoveryTime	Time at which the Details agent attempted to discover the device. This value is stored even if the device is not accessible using SNMP.	physicalChassis DiscoveryTime

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalChassis
The `physicalChassis` table stores the attributes of chassis entities.

fan

The `fan` view joins data from a number of tables and is equivalent to the `fan` table that existed in Network Manager versions 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `fan` view.

Table 504. fan view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	entityData entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalFan Model
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalFan RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalFan physicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalFan Name
entPhysicalDescr	A textual description of the entity. This value must include the full name and version identification of the system hardware type, software operating-system, and networking software.	entityData Description
serialNumber	The serial number of the entity.	physicalFan SerialNumber
modelName	Model name for this entity.	physicalFan Model

Table 504. fan view (continued)		
Column name	**Description**	**Containing table and field name**
isFieldReplaceable	Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False	physicalFan isFRU

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalFan
The `physicalFan` table represents fan cooling unit entities.

interface

The `interface` view joins data from a number of tables and is equivalent to the `interface` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `interface` view.

Table 505. interface view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	networkInterface entityData
ifIndex	The index of the interface.	networkInterface SNMPIndex
ifPhysAddress	The physical address of the interface.	networkInterface PhysicalAddress
ifName	The name assigned to the interface.	networkInterface ifName
ifDescr	A description of the interface.	networkInterface ifDescr
ifAlias	The alias for the interface.	networkInterface ifAlias
ifSpeed	An estimate of the current bandwidth of the interface in bits per second.	networkInterface ifSpeed

Table 505. interface view (continued)		
Column name	**Description**	**Containing table and field name**
ifHighSpeed	An estimate of the current bandwidth of the interface in units of 1,000,000 bits per second.	networkInterface ifHighSpeed
ifAdminStatus	The required state of the interface. Takes one of the following values: • Up • Down • Testing	networkInterface ifAdminStatus
ifOperStatus	The current operational state of the interface. Takes one of the following values: • Up • Down • Testing • unknown • dormant • notPresent • lowerLayerDown	networkInterface ifOperStatus
ifType	The interface type.	networkInterface IANAInterfaceType
ifTypeString	The textual string for the interface type.	networkInterface ifTypeString
ifMTU	The maximum transmission unit for this interface.	networkInterface MTU
ifPromiscuousMode	Indicates whether this interface only accepts packets or frames addressed to this station. Takes one of the following values: • True • False	networkInterface PromiscuousMode
ifConnectorPresent	Indicates whether the interface has a connector. Takes one of the following values: • True • False	networkInterface ConnectorPresent

Table 505. interface view (continued)

Column name	Description	Containing table and field name
accessIPAddress	The IP address through which this entity was discovered and will be monitored. **Note:** For non-IP entities, such as layer 1 optical devices, this field is null.	networkInterface accessIP
accessProtocol	An integer representation of the network protocol used by the presently-defined zone: • 0: Unknown • 1: IPv4 • 2: IPv4 that has been through network address translation (NAT) • 3: IPv6 • 4: Element Management System (EMS) key for a non-IP device	networkInterface accessProtocol
entPhysicalVendorType	The vendor-specific hardware type of this physical entity.	x_cdmPhysicalConnector Model
entPhysicalParentRelPos	The relative position of this entity within the containment.	x_cdmPhysicalConnector RelativePosition
entPhysicalIndex	The physical index for this entity.	x_cdmPhysicalConnector physicalIndex
entPhysicalName	The textual name of this physical entity.	x_cdmPhysicalConnector Name
entPhysicalDescr	The textual description of this physical entity.	x_cdmPhysicalConnector entPhysicalDescr
portNumber	The port number for this interface on the chassis device. The method of determining the port number is dependent on the make and model of the device that is discovered. For this reason, use this value with caution.	networkInterface portNumber
isTrunkPort	Indicates whether this physical interface is a VLAN trunk port.	networkInterface SwitchPortMode

Table 505. interface view (continued)		
Column name	**Description**	**Containing table and field name**
duplex	Actual duplex of the interface. Takes one of the following values: • HalfDuplex • FullDuplex • Auto • Unknown • Other	networkInterface OperationalDuplex

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

networkInterface
The `networkInterface` table represents interfaces on a chassis device.

module

The `module` view joins data from a number of tables and is equivalent to the `module` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `module` view.

Table 506. module view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalCard entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalCard Model
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalCard RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalCard Physical Index

Table 506. module view (continued)

Column name	Description	Containing table and field name
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalCard Name
entPhysicalDescr	A textual description of the entity. This value must include the full name and version identification of the system hardware type, software operating-system, and networking software.	entityData Description
serialNumber	The serial number of the entity.	physicalCard SerialNumber
modelName	Model name for this entity.	physicalCard Model
orderablePartNumber	Orderable part number for this entity.	physicalCard PartNumber
hardwareVersion	Hardware version for this entity.	physicalCard HWRevision
firmwareVersion	Firmware version for this entity.	physicalCard FWRevision
softwareVersion	Software revision.	physicalCard SWRevision
softwareImage		physicalCard softwareImage
isFieldReplaceable	Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False	physicalCard isFRU

Table 506. module view (continued)

Column name	Description	Containing table and field name
operStatus	Operational status of this card. Takes one of the following values: • unknown • ok • disabled • okButDiagFailed • boot • selfTest • failed • missing • mismatchWithParent • mismatchConfig • diagFailed • dormant • outOfServiceAdmin • outOfServiceEnvTemp	physicalCard operStatus
adminStatus	Administrative status of this card. Takes one of the following values: • unknown • enabled • disabled • reset • outOfServiceAdmin	physicalCard adminStatus
cardNumber	Indication of the relative position of this entity within the containment.	physicalCard RelativePosition

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalCard
The `physicalCard` table represents card entities.

other

The `other` view joins data from a number of tables and is equivalent to the `other` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `other` view.

Table 507. other view		
Column name	**Description**	**Containing table and field name**
entityId	The identifier of a network pipe entity from the `entityData` table.	physicalOther entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalOther vendorType
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalOther relativePosition
entPhysicalIndex	The physical index for this entity.	physicalOther physicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalOther name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData description
entPhysicalClass	Takes one of the following values: • 1: unknown • 2: other	physicalOther physicalClass

Related reference

entityData

The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalOther

The physicalOther table stores attributes of a component whose physical entity class is known, but does not match any of the supported values.

psu

The psu view joins data from a number of tables and is equivalent to the psu table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the psu view.

Table 508. psu view

Column name	Description	Containing table and field name
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalPowerSupply entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalPowerSupply Model
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalPowerSupply RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalPowerSupply PhysicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalPowerSupply Name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData Description

Table 508. psu view (continued)		
Column name	**Description**	**Containing table and field name**
serialNumber	The serial number of the entity.	physicalPowerSupply SerialNumber
modelName	Model name for this entity.	physicalPowerSupply Model
isFieldReplaceable	Indication of whether this piece of equipment is replaceable in the field. Takes one of the following values: • True • False	physicalPowerSupply isFRU
adminStatus	See the values for cefcFRUPowerAdminStatus in Eumerations table.	physicalPowerSupply powerAdminStatus
operStatus	See the values for cefcFRUPowerOperStatus in Eumerations table.	physicalPowerSupply powerOperStatus

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalPowerSupply
The `physicalPowerSupply` table represents a power supply unit (PSU) entity.

sensor

The `sensor` view joins data from a number of tables and is equivalent to the `sensor` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `sensor` view.

Table 509. sensor view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalSensor entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalSensor Model

Table 509. sensor view (continued)

Column name	Description	Containing table and field name
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalSensor RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalSensor physicalIndex
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalSensor Name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData Description

Table 509. sensor view (continued)

Column name	Description	Containing table and field name
sensorType	Sensor type. Takes one of the following values: • other • unknown • voltsAC • voltsDC • amperes • watts • hertz • celsius • percentRH • rpm • cmm • truthValue • specialEnum	physicalSensor sensorType
sensorScale	Sensor scale. Takes one of the following values: • unknown • yocto • zepto • atto • femto • pico • nano • micro • milli • Units • kilo • mega • giga • tera • exa • peta • zetta • yotta	physicalSensor sensorScale

Table 509. sensor view (continued)		
Column name	**Description**	**Containing table and field name**
sensorStatus	Sensor status. Takes one of the following values: • ok • unavailable • nonoperational • unknown	physicalSensor sensorStatus
sensorValue	The value for the sensor.	physicalSensor sensorValue

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalSensor
The `physicalSensor` table represents sensor entities.

slot

The `slot` view joins data from a number of tables and is equivalent to the `slot` table that existed in Network Manager version 3.9 and earlier , thereby ensuring backward compatibility.

The following table describes the `slot` view.

Table 510. slot view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	physicalSlot entityId
entPhysicalVendorType	An indication of the vendor-specific hardware type of the physical entity.	physicalSlot Model
entPhysicalParentRelPos	An indication of the relative position of this child component among all its sibling components. Sibling components are defined as entPhysicalEntries which share the same instance values of each of the entPhysicalContainedIn and entPhysicalClass objects.	physicalSlot RelativePosition
entPhysicalIndex	The physical index for this entity.	physicalSlot PhysicalIndex

Table 510. slot view (continued)		
Column name	**Description**	**Containing table and field name**
entPhysicalName	The textual name of the physical entity. The value of this object must be the name of the component as assigned by the local device and is suitable for use in commands entered at the console of the device. Depending on the physical component naming syntax of the device, this value might be a text name, for example *console*, or a single component number, for example a port number or a module number.	physicalSlot Name
entPhysicalDescr	A textual description of physical entity. This object must contain a string which identifies the manufacturer's name for the physical entity. The value must be set to a distinct value for each version or model of the physical entity.	entityData Description
powerRedundancyMode	Takes one of the following values: • unknown • notSupported • redundant • combined	physicalSlot SerialNumber

Related reference

entityData
The `entityData` table stores data on entities. This table belongs to the category *entities*.

physicalSlot
The `physicalSlot` table represents slot entities.

sourceEms

The `sourceEms` view joins data a number of tables. This view provides data on devices discovered by an EMS collector and provides a mapping between the device and the EMS or EMSs that manage the device.

The following table describes the `sourceEms` view.

Table 511. sourceEms view		
Column name	**Description**	**Containing table and field name**
entityId	Foreign key to the entityNameCache table. Must be unique for each entity across all domains.	discoverySource entityId

Table 511. sourceEms view (continued)		
Column name	**Description**	**Containing table and field name**
entityName	Name of the entity.	discoverySource entityName
source	Source of the data. This field takes one of the following values: • Unknown • Other • TopologyEditor • PresetLayer • Agent • Collector	discoverySource source
discoveryProtocol	Protocol of the data provided by this discovery source. This field takes one of the following values: • Unknown • Other • Manual • FlatFile • SNMP • Telnet • XML-RPC • VSphere • OtherJavaAPI • TL1 • CORBA	discoverySource discoveryProtocol
nativeId	Identifier used by the discovery source to identify a given device.	discoverySource nativeId
nativeIdString	String used by the discovery source to identify a given device.	discoverySource nativeIdString
emsEntityId	Automatically-incremented field that provides a unique value for each entity across all domains.	entityNameCache entityId
emsEntityName	Name of the entity.	entityNameCache entityName
emsName	Name of the EMS.	emsSystem emsName

Table 511. sourceEms view (continued)		
Column name	**Description**	**Containing table and field name**
domainMgrId	Automatically-incremented field that is unique for each domain.	domainMgr domainMgrId
domainName	Name of the domain.	domainMgr domainName

Common Data Model views

The Common Data Model (CDM) is an information model that provides consistent definitions for managed resources, business systems and processes, and other data, and the relationships between those elements. CDM is based on the unified modeling language (UML).

The IBM Common Data Model (CDM) overlay schema enables Network Manager to expose a subset of its data in a CDM-like relational representation corresponding to aspects of the CDM Computer System, Networking, Operating System, and Physical sub-models. The CDM schema complements the NCIM topology database by providing tables to allow the storage of extra CDM attributes.

This section describes the CDM views exposed by Network Manager. The CDM views have been defined using existing NCIM database tables and attributes.

CDM views

The CDM views are described below. For each CDM view the corresponding row lists the tables and views mapped to by the CDM view.

Table 512.	
CDM view	**Tables and views mapped to by this CDM view**
CDMCARD	This view maps to the following tables and views: • CDMMODLEOBJECT view • mappings table • module view • x_cdmCard table
CDMCHASSIS	This view maps to the following tables and views: • CDMMODELOBJECT view • chassis view • deviceFunction table • x_cdmChassis table
CDMCOMPUTER SYSTEM	This view maps to the following tables and views: • CDMMODELOBJECT view • chassis view • deviceFunction table • virtualMachine table • x_cdmComputerSystem table

Table 512. (continued)

CDM view	Tables and views mapped to by this CDM view
CDMFAN	This view maps to the following tables and views: • CDMMODELOBJECT view • fan view • mappings table • x_cdmFan table
CDMIPV4ADDRESS	This view maps to the following tables and views: • CDMMODELOBJECT view • ipEndPoint table
CDMIPV4NETWORK	This view maps to the following tables and views: • CDMMODELOBJECT view • subnet table
CDMIPV6ADDRESS	This view maps to the following tables and views: • CDMMODELOBJECT view • ipEndPoint table
CDMIPV6NETWORK	This view maps to the following tables and views: • CDMMODELOBJECT view • subnet table
CDMMODELOBJECT	Provides generally applicable data to the other CDM views. This view maps to the following tables and views: • entityData • domainMembers • domainMgr
CDMNETWORK INTERFACE	This view maps to the following tables and views: • CDMModelObject view • enumerations table • interface view • x_cdmNetworkInterface table
CDMOPERATING SYSTEM	This view maps to the following tables and views: • CDMModelObject view • chassis table • x_cdmOperatingSystem table

Table 512. (continued)

CDM view	Tables and views mapped to by this CDM view
CDMOTHER PHYSICALPACKAGE	This view maps to the following tables and views: • CDMMODELOBJECT view • chassis view • x_cdmOperatingSystem table
CDMPHYSICAL CONNECTOR	This view maps to the following tables and views: • CDMMODELOBJECT view • interface view • mappings table • x_cdmPhysicalConnector table
CDMPOWER SUPPLY	This view maps to the following tables and views: • CDMMODELOBJECT view • mappings table • psu view • x_cdmPowerSupply table
CDMSENSOR	This view maps to the following tables and views: • CDMMODELOBJECT view • mappings table • sensor view • x_cdmSensor table
CDMSLOT	This view maps to the following tables and views: • CDMMODELOBJECT view • mappings table • sensor view • x_cdmSensor table
CDMSNMPSYSTEM GROUP	This view maps to the following tables and views: • CDMMODELOBJECT view • chassis view

Related information

IBM Tivoli Common Data Model: Guide to Best PracticesThe Common Data Model (CDM) is an information model that provides consistent definitions for managed resources, business systems and processes, and other data, and the relationships between those elements. CDM is based on the unified modeling language (UML). This IBM Redpaper presents a set of example templates and scenarios that help you learn and apply the basics of the Common Data Model.

Chapter 24. Topology API reference

Read about the Topology API provided with Network Manager.

Overview of the Topology API

The Topology API is a RESTful interface that enables you to extract chassis device data from the Network Manager NCIM topology database into a JavaScript Object Notation (JSON) file that represents the device resources. This JSON file can be parsed and imported into a third-party tool, possibly together with data from other products, for presentation or further analysis.

In order to implement the Topology API within an integration you can use a command-line HTTP client that understands HTTPS, such as wget or curl, or whatever libraries your programming language of choice provides. A quick and simple way to test the output of the Topology API is to use GET and POST REST methods in a REST client within a web browser. Results are presented in JSON format.

Retrieving device data

You can use the Topology API to extract device data from the NCIM topology database.

Extracting topology data for all chassis devices

You can use the Topology API to extract topology data from the NCIM topology database for all chassis devices in the database. You could use this data to integrate with another product; however, this is not a complete database dump because it retrieves only chassis devices.

Syntax

Specify the following parameters within a REST client in order to extract topology data for all chassis devices.

Table 513. Parameters to extract topology data for all chassis devices

Parameter	Value
Method	GET

Table 513. Parameters to extract topology data for all chassis devices (continued)

Parameter	Value

URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/all?includeSubChassis=true|false`
	Where:
	• *HOST* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *ROOT-CONTEXT* is the path from the root of your file system to the Topology API.
	• `includeSubChassis` is an optional Boolean parameter. If this parameter is set to `true`, the results of the call include sub-chassis. If this parameter is set to `false`, the results of this call exclude sub-chassis. The default is `true`. For the purposes of this feature, a sub-chassis is defined as a device that exists in `ncim.chassis`, where its `entityId` in `ncim.entityData` is not equal to its `mainNodeEntityId`, and its `className` in `ncim.chassis` is equal to the `className` of the row for its `mainNodeEntityId`.
	For example:
	`https://myHost:16311/ibm/console/nm_rest/topology/devices/all?includeSubChassis=true`

Content Type	application/json

Extracting topology data for all chassis devices that belong to a set of classes

You can use the Topology API to extract topology data from the NCIM topology database for all chassis devices that belong to a specified set of classes, or that do not belong to a specified set of classes.

Syntax

Specify the following parameters within a REST client in order to extract topology data for all chassis devices that do or do not belong to a specified set of classes.

Table 514. Parameters to extract topology data for all chassis devices that do or do not belong to a specified set of classes

Parameter	Value
Method	GET

Table 514. Parameters to extract topology data for all chassis devices that do or do not belong to a specified set of classes (continued)

Parameter	Value

URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/classes?classes=CLASSNAME1,CLASSNAME2&include=true|false&includeSubChassis=true|false`
	Where:
	• *HOST* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *ROOT-CONTEXT* is the path from the root of your file system to the Topology API.
	• *classes* is a comma-separated list of class names. See "Retrieving class data" on page 789 for information about retrieving class data by using the Topology API.
	• `include` is an optional parameter. It can be set to `true`, in which case data is extracted for devices that belong to the specified classes, or `false`, in which case data is extracted for all devices except those that belong to the specified classes. The default is `true`.
	• `includeSubChassis` is an optional Boolean parameter. If this parameter is set to `true`, the results of the call include sub-chassis. If this parameter is set to `false`, the results of this call exclude sub-chassis. The default is `true`. For the purposes of this feature, a sub-chassis is defined as a device that exists in `ncim.chassis`, where its `entityId` in `ncim.entityData` is not equal to its `mainNodeEntityId`, and its `className` in `ncim.chassis` is equal to the `className` of the row for its `mainNodeEntityId`.
	For example:
	`https://myHost.com:16311/ibm/console/nm_rest/topology/devices/classes?classes=Router,Linux&includeSubChassis=true`

Content Type	application/json

Extracting topology data for all chassis devices within a specified network view

You can use the Topology API to extract topology data from the NCIM topology database for all chassis devices within a specified network view and all views under it, recursively. You could use this export to write new widgets for Network Manager. You could use this mode to integrate widgets from another product into a dashboard, as long as that product understands Network Manager entity identifiers or the other properties that the Topology API returns. If you are integrating with a third-party product, then that product does not need to know anything about network views; it just knows that it needs to call the Topology API with the view ID parameter, and display information for the devices that the API returns. In the case where you are using the network view tree to drive the Topology API export, this export would run interactively.

Syntax

Specify the following parameters within a REST client in order to extract topology data for all chassis devices within a specified network view.

Table 515. Parameters to extract topology data for all chassis devices within a specified network view	
Parameter	**Value**
Method	GET

URL	`https://host:port/root-context/nm_rest/topology/devices/viewId/view-id/allAttributes=true|false`
	Where:
	• *host* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *port* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *root-context* is the path from the root of your file system to the Topology API.
	• *view-id* is the numerical identifier of the network view from which you want to extract chassis device data.
	• Fix Pack 9 *allAttributes* is an optional parameter. If set to `true`, the query retrieves all attributes of the entities. If it is omitted, or set to `false`, the query retrieves only the entity IDs.
	For example:
	`https://myHost.com:16311/ibm/console/nm_rest/topology/devices/viewId/4203?allAttributes=true`

Content Type	application/json

Extracting topology data for all chassis devices within specified domains

You can use the Topology API to extract topology data from the NCIM topology database for all chassis devices within a specified domain or set of domains, or that do not belong to a specified set of domains.

Syntax

Specify the following parameters within a REST client in order to extract topology data for all chassis devices within a specified domain.

Table 516. Parameters to extract topology data for all chassis devices within a specified domain	
Parameter	**Value**
Method	GET

Table 516. Parameters to extract topology data for all chassis devices within a specified domain (continued)	
Parameter	**Value**

URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/domains?domains=DOMAIN_NAME&include=true|false`
	Where:
	• *HOST* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *ROOT-CONTEXT* is the path from the root of your file system to the Topology API.
	• *DOMAIN_NAME* is a comma-separated list of the names of the domains from which you want to extract chassis device data.
	• `include=true|false` is a domain filter. The filter considers the list of domains passed to the URL. If `include=true`, only devices in those domains are retrieved. If `include=false` the filter excludes all devices in those domains.
	For example:
	`https://myHost:16311/ibm/console/nm_rest/topology/devices/domains?domains=NCOMS1,NCOMS2,NCOMS3&include=true`

Content Type	application/json

Extracting topology data for a limited set of chassis devices

You can use the Topology API to extract topology data from the NCIM topology database for a limited set of chassis devices, such as a single device or a small number of devices.

Syntax

Specify the following parameters within a REST client in order to extract topology data for a limited set of chassis devices.

Table 517. Parameters to extract topology data for a limited set of chassis devices	
Parameter	**Value**
Method	POST
URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/entityIds` Where: • *HOST* is the hostname or IP address of the Dashboard Application Services Hub server. • *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311. • *ROOT-CONTEXT* is the path from the root of your file system to the Topology API.
Content Type	application/json

Table 517. Parameters to extract topology data for a limited set of chassis devices (continued)	
Parameter	**Value**
Payload	Comma-separated list representing the entity identifiers of the devices to retrieve; for example, `11649,11654,11661`

Retrieving entity identifiers for devices

In order to retrieve an entity identifier for devices for which you know the entity name, execute an SQL query similar to the following against the NCIM database:

```
select e.entityId from ncim.entityData e
inner join ncim.chassis c on e.entityId = c.entityId
where e.entityName = 'My Device';
```

In order to retrieve an entity identifier for devices for which you know the IP address, execute an SQL query similar to the following against the NCIM database:

```
select entityId from ncim.chassis
where accessIpAddress = '1.2.3.4';
```

For information on how to retrieve a list of chassis device or main node identifiers in a domain, see the *IBM Tivoli Network Manager Reference*.

Example JSON output for chassis devices

Use this information to understand the format of the JSON file that the Topology API produces as output.

The following code snippet shows an example of a JSON file produced as output for the following Topology API query. This query retrieves data for a single device with the entity identifier 1996.

```
https://myHost:16311/ibm/console/nm_rest/topology/devices/entityIds?id=1996
```

Example of JSON output

This query produced the following JSON output. For a description of this output, see the notes that follow the output.

Note: The actual output produced does not have the indentation in this example. The output is indented here in order to make the data easier to read.

```
{
    "devices":
    {
        "items":
        [
            {
                "memorysize":null,
                "classid":5,
                "classname":"NetworkDevice",
                "syslocation":"3rd Floor Lab",
                "entphysicalvendortype":"1.3.6.1.4.1.9.12.3.1.3.436",
                "manual":0,
                "entphysicalparentrelpos":-1,
                "accessipaddress":"172.30.233.101",
                "createtime":1376304329000,
                "entphysicalname":"2811 chassis",
                "cdmadminstate":0,
                "domainname":"PEMBERS",
                "alias":null,
                "entphysicalindex":1,
                "sysdescr":"Cisco IOS Software, 2800 Software (C2800NM-ADVIPSERVICESK9-M),
Version 12.4(24)T7, RELEASE SOFTWARE (fc2)\r\n
Technical Support: http://www.cisco.com/techsupport\r\n
Copyright (c) 1986-2012 by Cisco Systems, Inc.\r\n
```

```
Compiled Tue 28-Feb-12 10:43 by prod_rel_team",
                "sysname":"uk-t1-cj31.na.test.lab",
                "entitytype":1,
                "ostype":"Cisco IOS",
                "orderablepartnumber":"CISCO2811      ",
                "ipforwarding":"forwarding",
                "ifnumber":38,
                "sysobjectid":"1.3.6.1.4.1.9.1.576",
                "sysservices":"datalink(2) network(3) transport(4) application(7)",
                "entityname":"172.30.233.101",
                "domainmgrid":1,
                "hardwareversion":"V04 ",
                "displaylabel":"172.30.233.101",
                "syscontact":"someone@example.com",
                "osversion":"12.4(24)T7",
                "modelname":"CISCO2811      ",
                "sysuptime":449642945,
                "osimage":"flash:c2800nm-advipservicesk9-mz.124-24.T7.bin",
                "accessprotocol":"IPv4",
                "description":"Cisco IOS Software, 2800 Software (C2800NM-ADVIPSERVICESK9-M),
Version 12.4(24)T7, RELEASE SOFTWARE (fc2)\r\n
Technical Support: http://www.cisco.com/techsupport\r\n
Copyright (c) 1986-2012 by Cisco Systems, Inc.\r\n
Compiled Tue 28-Feb-12 10:43 by prod_rel_team",
                "discoverytime":1434637258000,
                "changetime":1376304329000,
                "serialnumber":"ZZ9PZA42",
                "entityid":1996,
                "entphysicaldescr":
"Cisco IOS Software, 2800 Software (C2800NM-ADVIPSERVICESK9-M),
Version 12.4(24)T7, RELEASE SOFTWARE (fc2)\r\n
Technical Support: http://www.cisco.com/techsupport\r\n
Copyright (c) 1986-2012 by Cisco Systems, Inc.\r\n
Compiled Tue 28-Feb-12 10:43 by prod_rel_team"
            }
        ],
        "properties":
        [
            {"entityid":"LONG"},
            {"classid":"LONG"},
            {"classname":"STRING"},
            {"entityname":"STRING"},
            {"createtime":"DATETIME"},
            {"changetime":"DATETIME"},
            {"displaylabel":"STRING"},
            {"description":"STRING"},
            {"entitytype":"LONG"},
            {"manual":"LONG"},
            {"alias":"STRING"},
            {"cdmadminstate":"LONG"},
            {"sysname":"STRING"},
            {"sysdescr":"STRING"},
            {"sysobjectid":"STRING"},
            {"syslocation":"STRING"},
            {"syscontact":"STRING"},
            {"sysuptime":"LONG"},
            {"sysservices":"STRING"},
            {"ifnumber":"LONG"},
            {"ipforwarding":"STRING"},
            {"entphysicalvendortype":"STRING"},
            {"entphysicalparentrelpos":"LONG"},
            {"entphysicalindex":"LONG"},
            {"entphysicalname":"STRING"},
            {"entphysicaldescr":"STRING"},
            {"serialnumber":"STRING"},
            {"modelname":"STRING"},
            {"orderablepartnumber":"STRING"},
            {"hardwareversion":"STRING"},
            {"ostype":"STRING"},
            {"osversion":"STRING"},
            {"osimage":"STRING"},
            {"accessipaddress":"STRING"},
            {"accessprotocol":"STRING"},
            {"memorysize":"LONG"},
            {"discoverytime":"DATETIME"},
            {"domainmgrid":"LONG"},
            {"domainname":"STRING"}
        ],
        "instrumentation":
        {
            "totalExecTime":142,
            "queryTime":105,
```

```
            "processTime":37,
            "count":1
        }
    }
}
```

Source of data in JSON output

The columns in the JSON output come from the following tables and views in the NCIM topology database:

1. chassis view: all of the columns come from this view, except for the columns listed in items 2 and 3.
2. domainMgr table: the following columns come from this table:
   - domainname
   - domainmgrid
3. entityClass table: the following columns come from this table:
   - classId
   - className
4. entityData table: the following columns come from this table:
   - alias
   - cdmadminstate
   - changetime
   - createtime
   - entitytype
   - entityname
   - displaylabel
   - description

For more information on these NCIM topology database tables and views, see the *IBM Tivoli Network Manager Reference*.

Structure of JSON output

The JSON output is structured as follows:

`devices:items`
An array containing one JSON object for each device that the query returns. The devices are not returned in any particular order.

`devices:properties`
An array that is returned once per query. This array contains JSON objects that specify the name and type of each property in the device instances contained in the `devices:items` arrray. Possible types are as follows:

DATETIME
Integer representing the number of milliseconds since midnight on 1st January 1970. No timezone conversion is performed on the value that is stored in the NCIM database.

LONG
64-bit signed integer. In order to minimize differences in the code between the Oracle and Db2, all integers returned by the Topology API are represented as 64-bit.

STRING
Quoted character string, UTF-8 encoded.

Note: A property whose value is NULL in the database is represented in the JSON output as the string `null`. If the property is a STRING type, the null does not have quotes around it. This distinguishes it from a string whose value is the four characters n, u, l, l.

devices:instrumentation
JSON object, returned once per query, and containing the following information:

count
Number of devices that the query returned.

queryTime
Number of milliseconds that the server took to execute the query on the NCIM database.

processTime
Number of milliseconds that the server took to process the result from the database and return it to the client.

totalExecTime
Number of milliseconds that passed between when the server received the query from the client and when it finished returning the result.

Fix Pack 7 Retrieving domain data

You can use the Topology API to extract a list of the domains defined in the `ncim.domainMgr` NCIM database table.

Syntax

Specify the following parameters within a REST client.

Table 518. Parameters to retrieve data for all domains

Parameter	Value
Method	GET

URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/meta/domains`
	Where:
	• *HOST* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *ROOT-CONTEXT* is the path from the root of your file system to the Topology API. This path is usually `ibm/console`.
	For example:
	`https://myHost:16311/ibm/console/nm_rest/topology/devices/meta/domains`

Content Type	application/json

Example JSON output for domains

Use this information to understand the format of the JSON file that the Topology API produces as output.

The following code snippet shows an example of a JSON file produced as output for the following Topology API query.

```
https://myHost:16311/ibm/console/nm_rest/topology/
devices/meta/domains
```

Example of JSON output

This query produced the following JSON output. For a description of this output, see the notes that follow the output.

Note: The actual output produced does not have the indentation in this example. The output is indented here in order to make the data easier to read.

```
{
  "devices":
  {
    "instrumentation":
    {
      "count":1,
      "queryTime":4,
      "totalExecTime":8,
      "processTime":4
    },
    "items":
    [
      {
        "lastUpdated":1552404469000,
        "webtopDataSource":"NCO_AGG_P",
        "creationTime":1546622278000,
        "domainName":"NCOMS",
        "description":null,
        "domainHost":"10.10.10.232",
        "domainMgrId":1,
        "managerName":"PrecisionIP",
        "domainPort":7968,
        "batchUpdatePercent":0
      }
    ],
    "properties":
    [
      {"domainmgrid":"LONG"},
      {"domainname":"STRING"},
      {"creationtime":"DATETIME"},
      {"lastupdated":"DATETIME"},
      {"managername":"LONG"},
      {"description":"STRING"},
      {"webtopdatasource":"STRING"},
      {"domainhost":"STRING"},
      {"domainport":"LONG"},
      {"batchupdatepercent":"LONG"}
    ]
  }
}
```

Source of data in JSON output

The columns in the JSON output come from the `domainMgr` table in the NCIM topology database. For more information on these NCIM topology database tables and views, see the *IBM Tivoli Network Manager Reference*.

Structure of JSON output

The JSON output is structured as follows:

`devices:items`
An array containing one JSON object for each device that the query returns. The devices are not returned in any particular order.

`devices:properties`
An array that is returned once per query. This array contains JSON objects that specify the name and type of each property in the device instances contained in the `devices:items` arrray. Possible types are as follows:

DATETIME
Integer representing the number of milliseconds since midnight on 1st January 1970. No timezone conversion is performed on the value that is stored in the NCIM database.

LONG
64-bit signed integer. In order to minimize differences in the code between the Oracle and Db2, all integers returned by the Topology API are represented as 64-bit.

STRING
Quoted character string, UTF-8 encoded.

Note: A property whose value is NULL in the database is represented in the JSON output as the string `null`. If the property is a STRING type, the null does not have quotes around it. This distinguishes it from a string whose value is the four characters n, u, l, l.

`devices:instrumentation`
JSON object, returned once per query, and containing the following information:

`count`
Number of devices that the query returned.

queryTime
Number of milliseconds that the server took to execute the query on the NCIM database.

processTime
Number of milliseconds that the server took to process the result from the database and return it to the client.

totalExecTime
Number of milliseconds that passed between when the server received the query from the client and when it finished returning the result.

Fix Pack 7 Retrieving class data

You can use the Topology API to extract a list of the classes that devices in the topology belong to.

Syntax

Specify the following parameters within a REST client.

Table 519. Parameters to retrieve data for all classes

Parameter	Value
Method	GET

URL	`https://HOST:PORT/ROOT-CONTEXT/nm_rest/topology/devices/meta/classes`
	Where:
	• *HOST* is the hostname or IP address of the Dashboard Application Services Hub server.
	• *PORT* is the secure port number of the Dashboard Application Services Hub server. By default, this is 16311.
	• *ROOT-CONTEXT* is the path from the root of your file system to the Topology API. This path is usually `ibm/console`.
	For example:
	`https://myHost:16311/ibm/console/nm_rest/topology/devices/meta/classes`

Content Type	application/json

Example JSON output for classes

Use this information to understand the format of the JSON file that the Topology API produces as output.

The following code snippet shows an example of a JSON file produced as output for the following Topology API query.

```
https://myHost:16311/ibm/console/nm_rest/topology/
devices/meta/classes
```

Example of JSON output

This query produced the following JSON output. For a description of this output, see the notes that follow the output.

Note: The actual output produced does not have the indentation in this example. The output is indented here in order to make the data easier to read.

```
{
  "devices":
  {
    "instrumentation":
    {
       "count":427,
       "queryTime":15,
       "totalExecTime":20,
       "processTime":5
    },
    "items":
    [
      {"classId":6,"className":"3Com","superClassId":5,"managerName":
"PrecisionIP","classType":"NetworkDevice"},
      {"classId":7,"className":"3ComAccessPoint","superClassId":6,"managerName"
:"PrecisionIP","classType":"Router"},
      {"classId":8,"className":"3ComCoreBuilder","superClassId":6,"managerName"
:"PrecisionIP","classType":"Switch"},
      ...
      {"classId":161,"className":"zOS","superClassId":200,"managerName":
"PrecisionIP","classType":"EndNode"}
    ],
    "properties":
    [
      {"classid":"LONG"},
      {"classname":"STRING"},
      {"superclassid":"LONG"},
      {"classtype":"STRING"},
      {"managername":"STRING"}
    ]
  }
}
```

Source of data in JSON output

The columns in the JSON output come from the `entityClass` table in the NCIM topology database. For more information on these NCIM topology database tables and views, see the *IBM Tivoli Network Manager Reference*.

Structure of JSON output

The JSON output is structured as follows:

`devices:items`
An array containing one JSON object for each device that the query returns. The devices are not returned in any particular order.

`devices:properties`
An array that is returned once per query. This array contains JSON objects that specify the name and type of each property in the device instances contained in the `devices:items` arrray. Possible types are as follows:

DATETIME
Integer representing the number of milliseconds since midnight on 1st January 1970. No timezone conversion is performed on the value that is stored in the NCIM database.

LONG
64-bit signed integer. In order to minimize differences in the code between the Oracle and Db2, all integers returned by the Topology API are represented as 64-bit.

STRING
Quoted character string, UTF-8 encoded.

Note: A property whose value is NULL in the database is represented in the JSON output as the string `null`. If the property is a STRING type, the null does not have quotes around it. This distinguishes it from a string whose value is the four characters n, u, l, l.

`devices:instrumentation`
JSON object, returned once per query, and containing the following information:

`count`
Number of devices that the query returned.

queryTime
Number of milliseconds that the server took to execute the query on the NCIM database.

processTime
Number of milliseconds that the server took to process the result from the database and return it to the client.

totalExecTime
Number of milliseconds that passed between when the server received the query from the client and when it finished returning the result.

Part 4. Discovery reference

To make advanced changes to the discovery, first understand the different aspects of discovery, including the discovery processes, phases, stages, Helpers, agents, stitchers and traps.

Note: The Discovery databases are not described in this section; they are described in the Management databases node.

Chapter 25. Discovery process

The Network Manager discovery process produces a network topology that includes connectivity and containment data.

Discovery subprocesses

The discovery process consists of several subprocesses that work together to discover devices and device interconnectivity.

When you launch a discovery, the internal Network Manager discovery engine (`ncp_disco`) is run. The ncp_disco engine manages the process of discovering device existence and interconnectivity.

Whenever you launch a full discovery the Discovery Engine, ncp_disco, rereads its configuration files. The Discovery Engine also instructs the Helper Server and the individual helpers to reread their configuration files. This is controlled by the `DiscoReadConfig()` rule within the FullDiscovery stitcher file.

Note: When you launch a partial discovery, ncp_disco does not read its configuration files.

The Discovery engine operates by detecting the existence of a device on the network and querying the device for inventory and connectivity information, which is subsequently processed or 'stitched' together to generate a connectivity or topology model. The discovery engine components are described in Table 520 on page 795.

Table 520. Discovery components

Name	Description
Finders	Finders discover the existence of devices but do not retrieve connectivity information.
Agents	ncp_disco uses discovery agents to request connectivity information from devices that the finders have discovered. There are a variety of agents, each specialized to retrieve information from different devices, and, in certain cases, to use different protocols. Agents do not have any direct interaction with the network, but instead retrieve information through the Helper Server. Agents can be libraries or text files, and are specialized for particular protocols, devices or classes. The Discovery engine starts each agent as a dependent unmanaged process. In the event of a failure of the Discovery engine, the Process controller, ncp_ctrl, shuts down each of the agents started by the discovery.
Helper Server	The Helper Server manages the helpers and stores the information that is retrieved from the network. Discovery agents retrieve their information through the Helper Server to reduce the load on the network. The Helper Server can service the requests directly with cached data or pass on the request to the appropriate helper.
Helpers	The helpers retrieve information from the network on behalf of the discovery agents. Helpers also translate agent queries into the appropriate network protocol and make requests to the devices.
Stitchers	Stitchers are processes that transfer, manipulate and distribute data between databases. The discovery stitchers are also responsible for processing the information collected by the agents and using this information to create the network topology. A predefined set of stitchers is included with Network Manager. You can modify existing stitchers or write new stitchers to perform custom manipulation of your network topology. For example, you can write a stitcher to make your device interfaces appear with a custom naming convention. Stitchers are coded using the stitcher language.

Discovery timing

Each full discovery consists of one or more discovery cycles. The division of a full discovery into multiple discovery cycles enables the discovery to complete in a timely way.

In the first discovery cycle, Network Manager discovers the existence of a predetermined majority of devices on the network, and proceeds to complete all data collection and processing operations associated with these devices. When Network Manager has discovered the existence of a predetermined majority of devices on the network, Network Manager enters the *blackout state*.

Any devices that Network Manager discovers during the blackout state are placed into a database table named finders.pending. These devices are only processed in the following discovery cycle. This means that the discovery process does not have to wait for all devices to be discovered before proceeding to the more detailed data collection and data processing operations.

Note: Ideally a discovery should complete in a single discovery cycle; however, sometimes it is not possible to discover the existence of entities sufficiently quickly as a result more discovery cycles are needed. Reasons why the system does not discover the existence of entities sufficiently quickly include: ping sweeping of sparsely populated subnets, and lack of access to devices. First-time discoveries often have multiple cycles. This can be mitigated by using the BuildSeedList.pl script to build a seed list after the initial discoveries. This seed list will then be used in subsequent discoveries to find devices in a more timely manner.

By default, each discovery cycle is made up of a data collection stage and a data processing stage. The data collection stage is in turn broken up into three phases. Figure 30 on page 796 shows a timing diagram for a discovery that requires two discovery cycles to complete.

The data collection and data processing stages are briefly described in Table 521 on page 797.

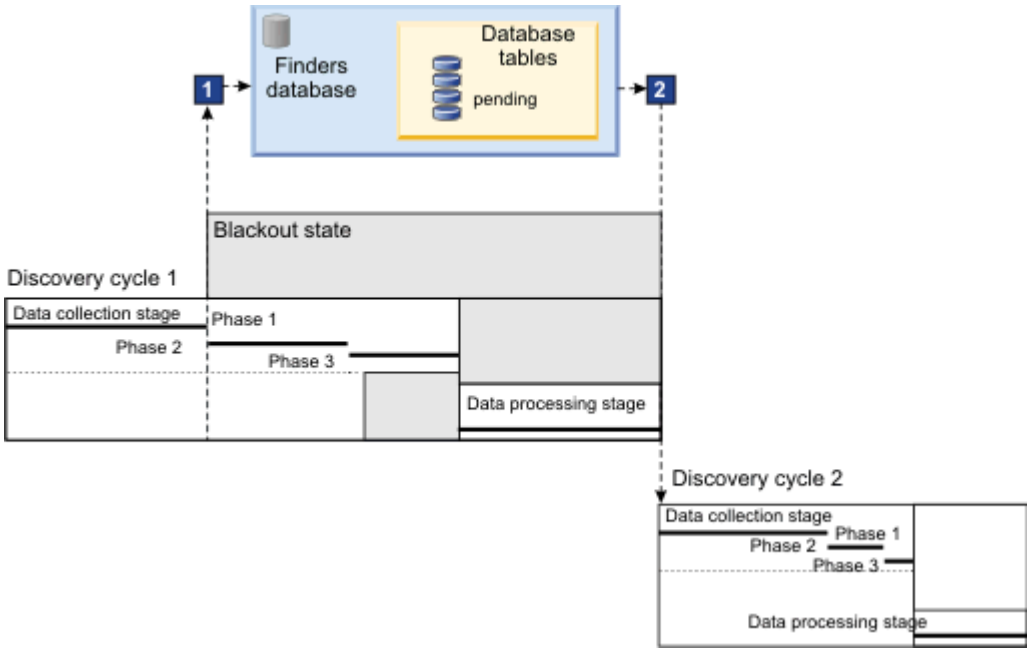


Figure 30. Discovery timing for a full discovery with two discovery cycles

In Figure 30 on page 796, the blackout state for the first discovery cycle begins and ends at the instants indicated by the numbers 1 and 2 respectively:

1: *Blackout state begins*. A predetermined majority of devices on the network have now been discovered. Any devices discovered after this point are placed into the finders.pending table for processing in the subsequent discovery cycle.

2: *Blackout state ends*. Devices stored in the finders.pending table are now processed in the subsequent discovery cycle.

Note: If the network being discovered is particularly large or complex, more than two discovery cycles may be required to complete a full discovery. In this case, each discovery cycle, except for the last cycle, has its own blackout state.

Table 521. Data collection and data processing stages

Stage or Phase	Description
Data collection stage	During this stage, Network Manager interrogates the network for device information, using the finder, agent and helper components of DISCO. The data collection stage is divided into three phases, which are described in this table.
Data collection: first phase	During this phase, finders identify devices on the network. Phase one completes when the device *find rate* drops below a certain level. For each device discovered, agents retrieve device details, IP addresses associated with the device, and device connectivity information.
Data collection: second phase	During this phase, an agent retrieves IP address to MAC address mapping data.
Data collection: third phase	During this phase agents download all forward database table information for the network switches and ping all devices to confirm the accuracy of the contents of the forward database tables.
Data processing stage	During this stage, Network Manager deduces the network topology based on data collected during the data collection stage. Stitchers analyze the data collected and build a network topology that includes connectivity and containment data.

Discovery stages and phases

The discovery process can be divided into two stages: data collection and data processing. The stages are subdivided into phases.

Data processing stage

Topology deduction takes place during the data processing stage, as the information from the data collection stage is analyzed, interpreted and processed by the stitchers. The culmination of the data processing stage is the production of the containment model.

The data processing stage corresponds to creating the topology. This is the final conceptual step in the discovery cycle.

The data processing and data collection stages usually overlap, because you can configure the stitchers to begin processing connectivity information from different discovery agents before the main stitching operation begins.

Data collection stage

The data collection stage involves interrogating the network for device information to produce a network topology. DISCO uses the finders, agents and helpers during the data collection stage. The data collection stage can be further subdivided into a number of phases.

First phase

In the first phase of data collection, the finders identify all the devices that exist on the network. Generally, a phase can be completed when all the launched processes have completed their operation. However, although you might want to wait until all devices have been discovered by the finders before proceeding to phase two, it is inefficient to hold back the discovery process by waiting indefinitely. The

first phase therefore completes when the *find rate* drops below a certain level, determined by no devices being discovered for the amount of time specified in `disco.config.m_NothingFndPeriod`.

The following conceptual steps in the discovery cycle take place during data collection phase one:

- Discovering device existence
- Discovering device details (standard)
- Discovering associated device addresses
- Discovering device connectivity

Agents in the first phase

Some agents return data that can be used to find other devices, for example, the IP address of remote neighbors, or the subnet within which a local neighbor exists. This mechanism is known as *feedback*.

The Feedback stitcher manages feedback by sending the information returned by the agents to the Ping finder for inclusion in the discovery. However, the blackout state ensures that any agent involved in the feedback process must be run in phase one for devices to be discovered in the current discovery cycle.

Phase one also usually involves the Switch discovery agents downloading all VLAN and interface information.

Blackout state

After phase one, the discovery enters the *blackout state*. The finders have discovered the existence of a pre-determined majority of devices on the network. Any new device addresses discovered in the blackout state, either by the finders or recursively by a discovery agent, are put into the finders.pending database table.

Devices in the finders.pending database table are processed in the next discovery. If there are devices in the finders.pending database table, the next discovery starts as soon as the current discovery finishes.

Second phase

After the criteria for the completion of phase one have been fulfilled, phase two begins. To map layers two and three of the OSI model, the ARP Cache discovery agent populates the Helper Server with ARP data, which is a list of device IP address-to-MAC address resolution.

Before the discovery can transfer from phase two to phase three, the processes from phase two must have completed their operation. An agent is considered to have finished after all entities in its despatch table are also in its returns table.

The agents are multithread, and records of discovered devices passed to the agents are tagged with a certain phase. Consequently, at any time an agent can be processing devices in two separate phases. If any action that should have occurred in phase two is detected after phase three has begun, phase three continues while the agent runs through phase two processing.

Third phase

By phase three, the discovery process has full knowledge of the devices that exist within the network (acquired from phase one) and access to full IP address-to-MAC address mappings for all devices in the Helper Server (acquired from phase two). The Switch agents can now proceed to download all the forward database table information of the network switches whilst pinging all devices to confirm the accuracy of the contents of the forward database tables.

When phase three has finished, which is signified by the completion of all processes scheduled to run in the phase, the discovery is ready to proceed from the data collection stage to the data processing stage, where all the connectivity information is knitted together to form a network topology.

Impact of the stages and phases approach on DISCO processes

The division of the data collection stage into phases affects all the processes involved in the discovery and network topology deduction, because the phases are processed in order. Any given phase cannot begin until the criteria for completion of the previous phase have been met.

All the processes of DISCO must therefore have an associated phase (or phases) in which they are allowed to operate. Thus, whilst the finders are typically configured to run through all phases, you might want to configure certain discovery agents to operate only within a specific phase(s). The flexibility of DISCO allows you to have processes that are intelligent enough to behave differently when they operate within different phases, and can pass control to other processes or stop operation until the start of their next operational phase.

Advantages of staged discovery

There are several reasons why it is advantageous to apply a staged and phased approach to discovery.

Switch connectivity

In determining the connectivity of some devices, it is sometimes necessary for the discovery agent to know all the devices that exist before requesting particular Management Information Base (MIB) variable(s), especially if the requested information is transient.

An example is when the layer 2 agents discover connectivity between Ethernet switches. Ethernet switches have forward database tables that expire over time. So, to ensure that a switch has a fully populated forward database table at the time of interrogation, you could ping all devices associated with the switch.

You would therefore configure the switch discovery agents to perform some other processing in data collection phase one. After the agents receive the signal that phase one has been completed (that is, all devices have been found) they can start phase two operations. For example, they could ping all devices within the discovery domain while downloading the forward database tables for all switches.

Mapping subnet boundaries

One limitation of configuring individual discovery agents to make individual ARP requests directly from the Helper Server is that the ARP helper cannot run simultaneously on multiple subnets unless it is specifically configured to do so. To resolve this problem, use a special ARP Cache discovery agent that imitates a generic discovery agent (in the sense that entities can be sent to it) but that also can map boundaries or different layers of the OSI model.

The ARP Cache discovery agent can inquire about ARP caches that exist on routers. It uses this information to populate the ARP helper database within the Helper Server and build up full device IP address to MAC address mapping without having to rely on the ARP helper.

This approach can be applied when using switch discovery agents that need to perform IP address-to-MAC address resolution before they can start operation. Following the example above, you could configure your discovery data collection stage to have three phases:

- Phase one: Find all devices that exist on the network.
- Phase two: Use the ARP Cache discovery agent to populate the Helper Server with full IP address to MAC address mappings.
- Phase three: Ping all devices and invoke the switch discovery agents by downloading the forward database tables for all switches in the network, using the IP address to MAC address mappings determined in phase two.

Multiphase discovery agents

Another possible consequence of dividing the data collection stage into phases is that you can configure the discovery agents to perform different operations within different phases.

Although a discovery agent is programmed to start operating in phase two, it could also conduct some other operation in phase one. This is because the end of phase one signifies only that all devices have been discovered. The agent could be configured to perform other actions such as downloading interfaces, issuing Telnet requests, or downloading other MIB variables during phase one. Only after phase two has started does the agent begin to process instructions specific to phase two.

Tip: It is good practice to configure the discovery to occur over multiple phases, to ensure maximum accuracy of the deduced topology.

Effect of discovery multiphasing on network traffic

One of the main benefits of multiphasing is reduced network traffic.

Because similar types of network requests are grouped in phases, data can be cached in the Helper Server to reduce the network load. The Helper Server is the intermediary between the discovery agents and the network, and can amalgamate multiple pings of the same device into one block so that they are resolved into a single ping.

The Helper Server also has a request pool that ensures that the Helper Server does not overload the network. The request pool does this by restricting the number of simultaneously-handled requests.

Criteria for multiphasing

The main criterion for configuring a discovery that has multiple phases is to assess the requirements of the different operations that need to be performed during the discovery process. For example, Ethernet-based discovery agents require at least two phases. It is possible to have discovery agents that can operate in any phase.

Managing the phases

The different phases of the discovery data collection stage are managed by an internal *phase manager*.

The phase manager:

- Reads the maximum overall phase number and calculates the total number of phases when all the discovery agent and stitcher definition files are loaded.
- Calculates the phase and process dependencies, that is, which discovery agents are scheduled to run in which phases.
- Monitors the processes running during the phases.

When the phase manager detects that all the processes for the current phase have completed, it sends a signal indicating phase completion for all the processes that are waiting to be launched in the next phase.

Discovery cycles

A discovery cycle has occurred when the discovery data flow for a particular cycle has gone from start to finish. A full discovery might require more than one cycle.

The discovery data flow can be categorized into the following conceptual steps:

- Discovering device existence
- Discovering device details (standard)
- Discovering device details (context-sensitive)
- Discovering associated device addresses
- Discovering device connectivity
- Creating the topology

These steps follow the discovery data flow in order from start to finish, with the exception of discovering device details (context-sensitive), which replaces discovering device details (standard) if the discovery is context-sensitive.

Discovering device existence

The discovery of device existence is carried out in several steps.

Figure 31 on page 801 shows how the initial existence of devices on the network is discovered.

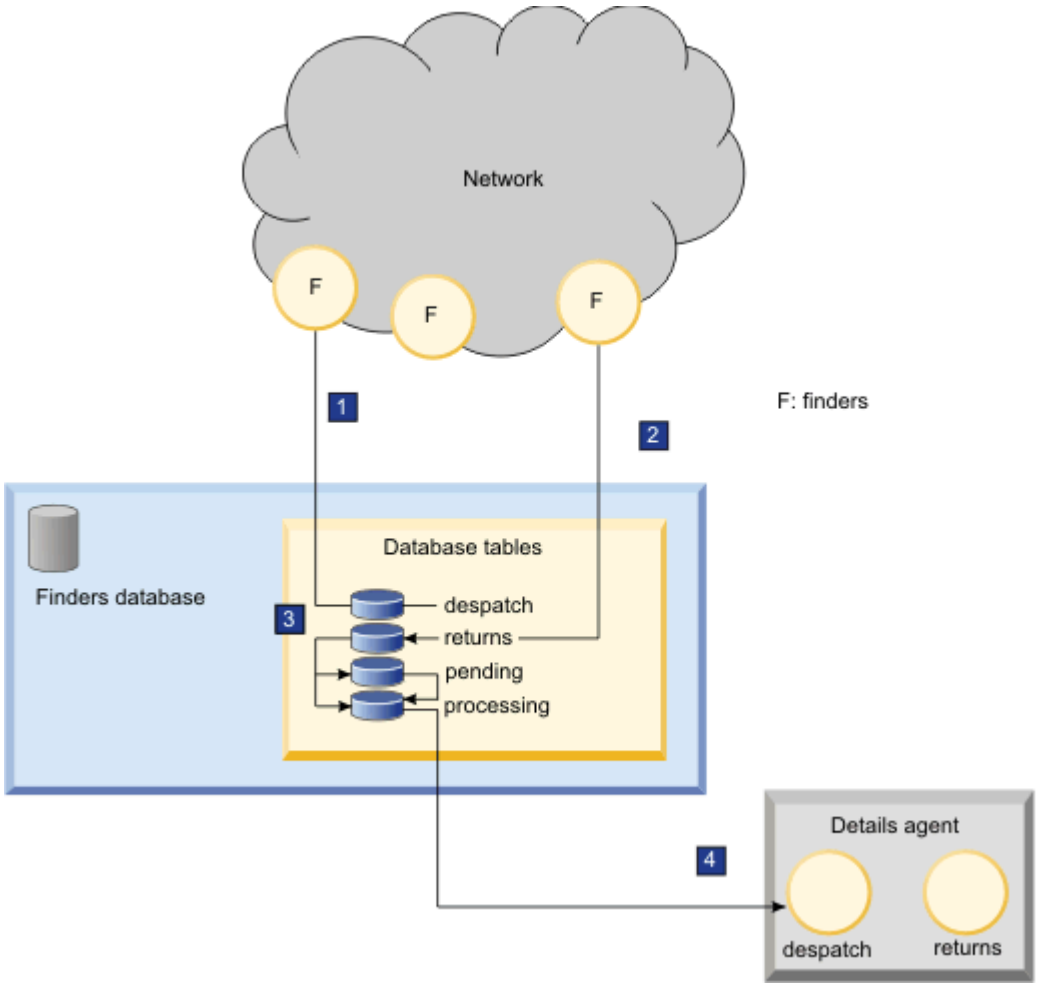


Figure 31. Discovery process flow: device existence

The process flow shown in Figure 31 on page 801 is described below.

1: The finders receive their instructions from their configuration files and the inserts made into the finders.despatch table, then proceed to the network to look for devices.

2: The finders return the device existence information to the finders.returns table.

3: After the device existence information is placed into the finders.returns table, a stitcher moves the information to the finders.processing table. This signifies that the network entity is being processed by DISCO. If the discovery is in the blackout state, the information is placed into the finders.pending table instead.

4: A stitcher moves the information about device existence from the finders.processing table to the Details.despatch table, ready for processing by the Details agent.

Discovering device details (standard)

The standard discovery of device details is carried out in several steps.

Figure 32 on page 802 shows how device details are discovered in a standard discovery.

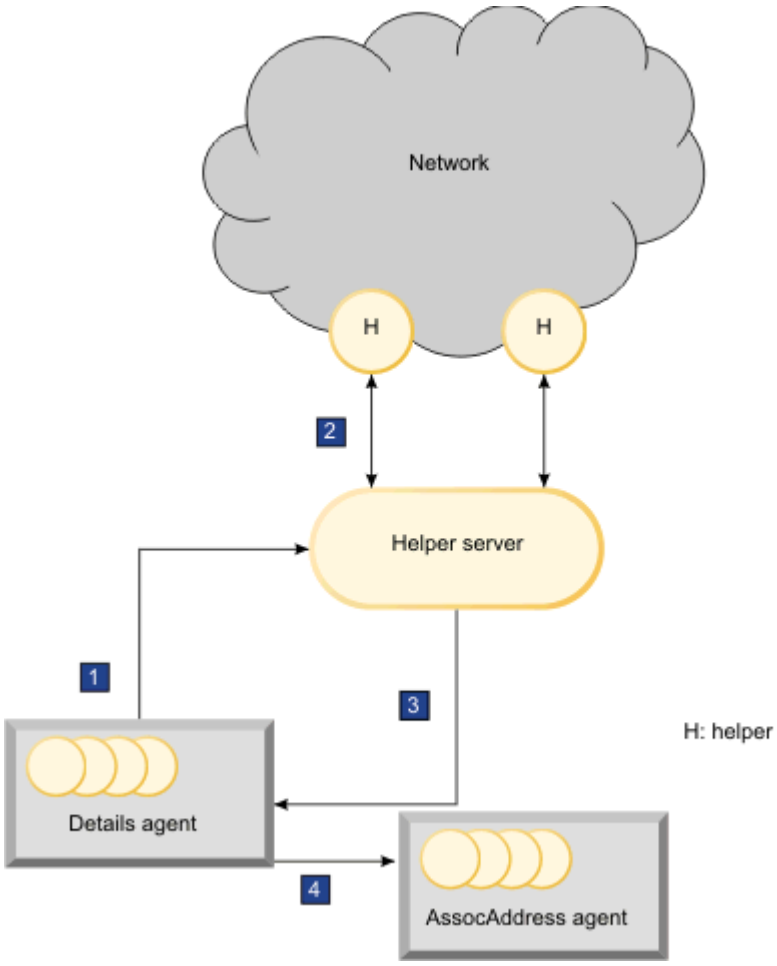


Figure 32. Discovery Process Flow: Device Details (Standard)

The process flow shown in Figure 32 on page 802 is described below.

1: All the agent despatch tables are active, so an insertion into the Details.despatch table automatically triggers the Details agent to discover basic device information and determine whether SNMP access to the device is available.

2: The Details agent interrogates the network through the Helper Server. Requests are cached to reduce the number of times that the helpers (represented by the letter H in Figure 32 on page 802) must interrogate the network directly.

3: The information retrieved from the network is returned to the Details.returns table.

4: The information in the Details.returns table is passed to the despatch table of the Associated Address (AssocAddress) agent for processing.

Discovering device details (context-sensitive)

The discovery of context-sensitive device details is carried out in several steps.

Figure 33 on page 803 shows how device details are discovered in a context-sensitive discovery.

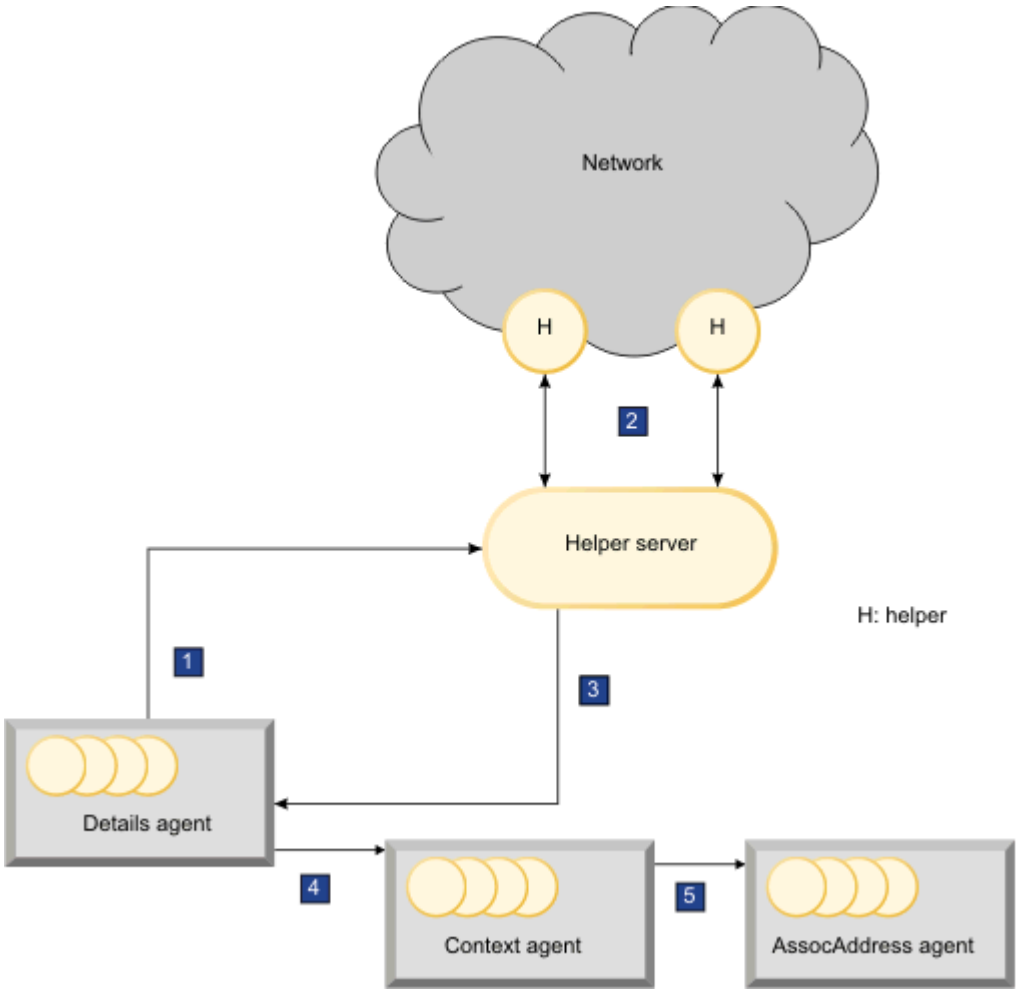


Figure 33. Discovery process flow: device details (context-sensitive)

The process flow shown in Figure 33 on page 803 is described below.

1: All the agent despatch tables are active, so an insertion into the Details.despatch table automatically triggers the Details agent to discover basic device information and determine whether or not SNMP access to the device is available.

2: The Details agent interrogates the network through the Helper Server. Requests are cached to reduce the number of times that the helpers must interrogate the network directly.

3: The information retrieved from the network is returned to the Details.returns table.

4: The information in the Details.returns table is passed to the despatch table of the appropriate Context agent, which adds context tags.

5: After the Context agent has finished its processing, the information is passed to the despatch table of the Associated Address (AssocAddress) agent for processing.

Discovering associated device addresses

There are several steps in the process flow during the discovery of associated device addresses.

The following figure shows how associated device addresses are discovered.

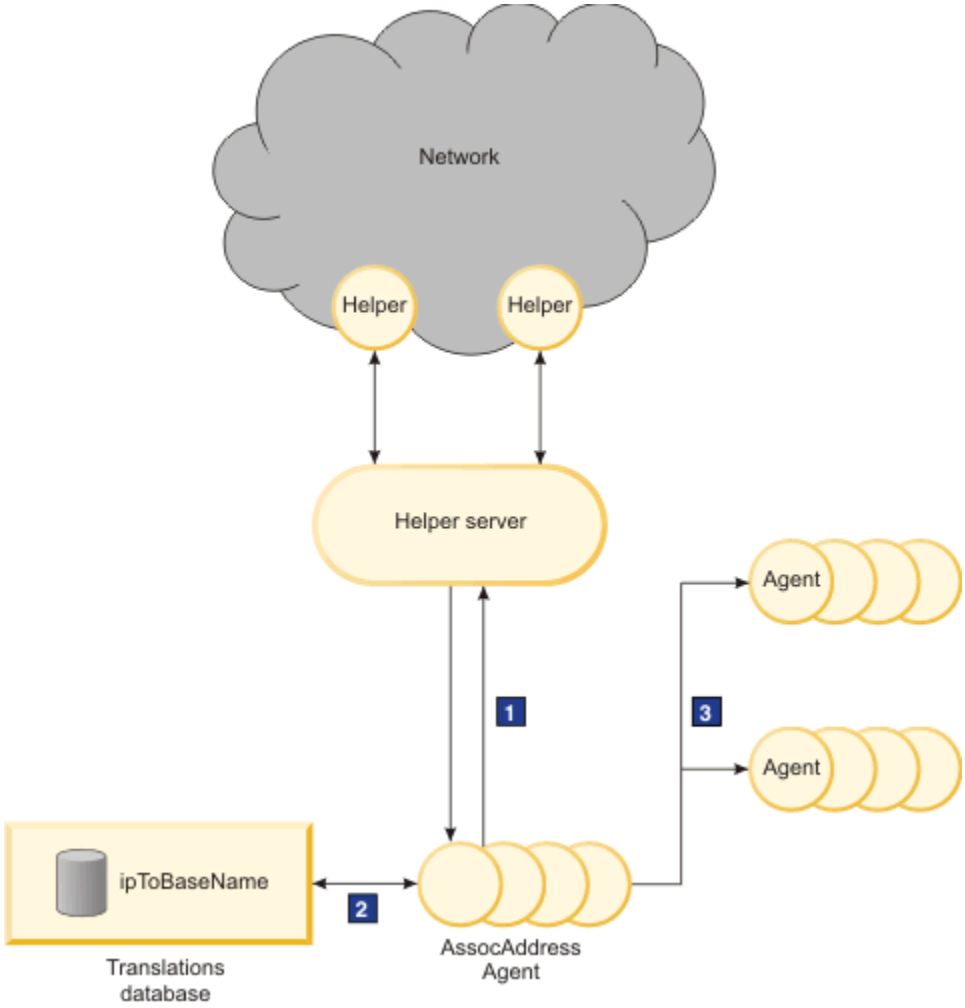


Figure 34. Discovery process flow: associated device addresses

The following process flow describes Figure 34 on page 804:

1: The Associated Address agent uses the Helper Server to download all the IP addresses associated with the interfaces of the device that is under investigation.

2: The Associated Address agent checks the IP addresses against the registry of addresses, the translations.ipToBaseName table. The details are also added to this registry. If the device has already been discovered by another of its addresses (that is, if the translations.ipToBaseName table already contains a record for this device), the details of the device are not sent to the discovery agents.

3: Provided the device has not already been discovered, the stitchers pass the details to the appropriate discovery agents, as specified in the `DiscoAgents.cfg` configuration file.

Discovering device connectivity

The discovery of device connectivity is carried out in several steps.

The following figure shows how device connectivity is discovered, as well as how devices are discovered recursively.

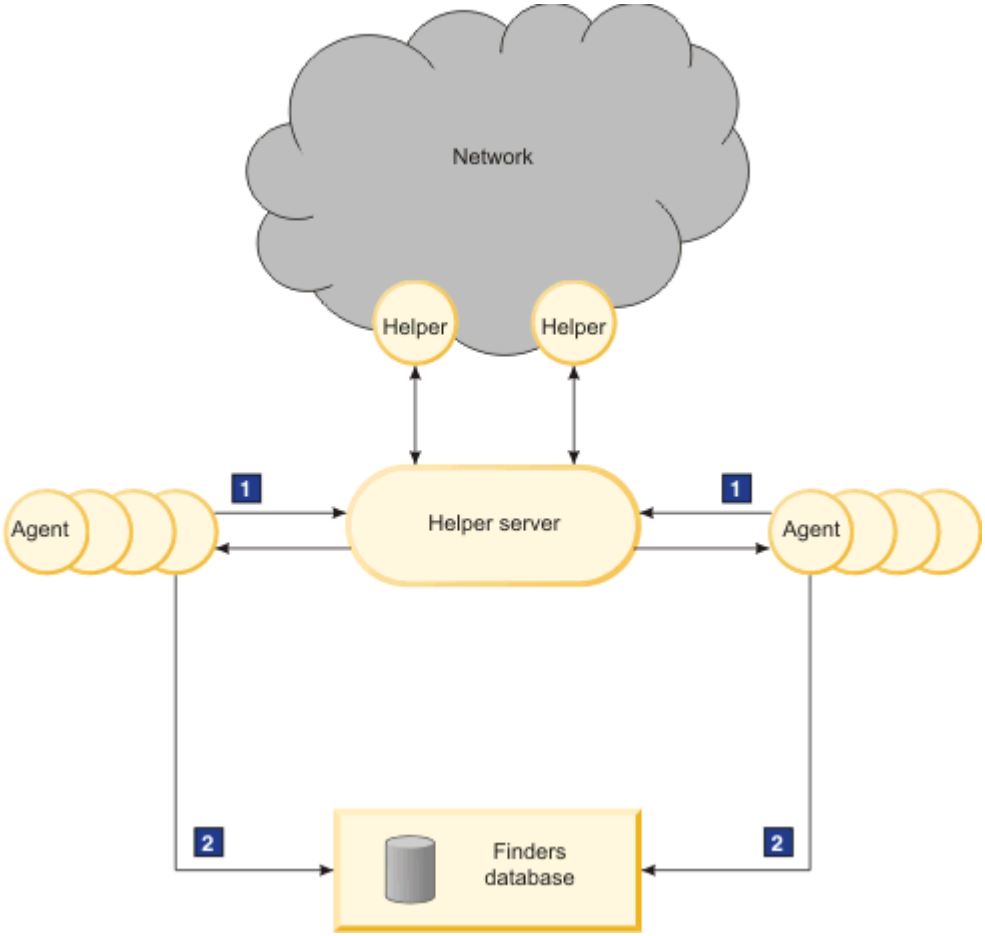


Figure 35. Discovery process flow: device connectivity

The following process flow describes Figure 35 on page 805:

1: When information is inserted into the `despatch` table of a discovery agent, the agent attempts to discover the connectivity information for that device. The agent sets up a TCP socket-based communication link with the Helper Server and requests the appropriate connectivity information.

2: A stitcher passes the addresses of the remote neighbors of the device, and the subnet address or addresses of the device, to a finder for discovery. Because these addresses might not exist, and also might not be in the specified discovery scope, the addresses must run through the discovery process from the beginning.

Creating the topology

The creation of the topology is carried out in several steps.

The following figure shows a simplified data flow for the creation of the topology from the raw data returned by the discovery agents

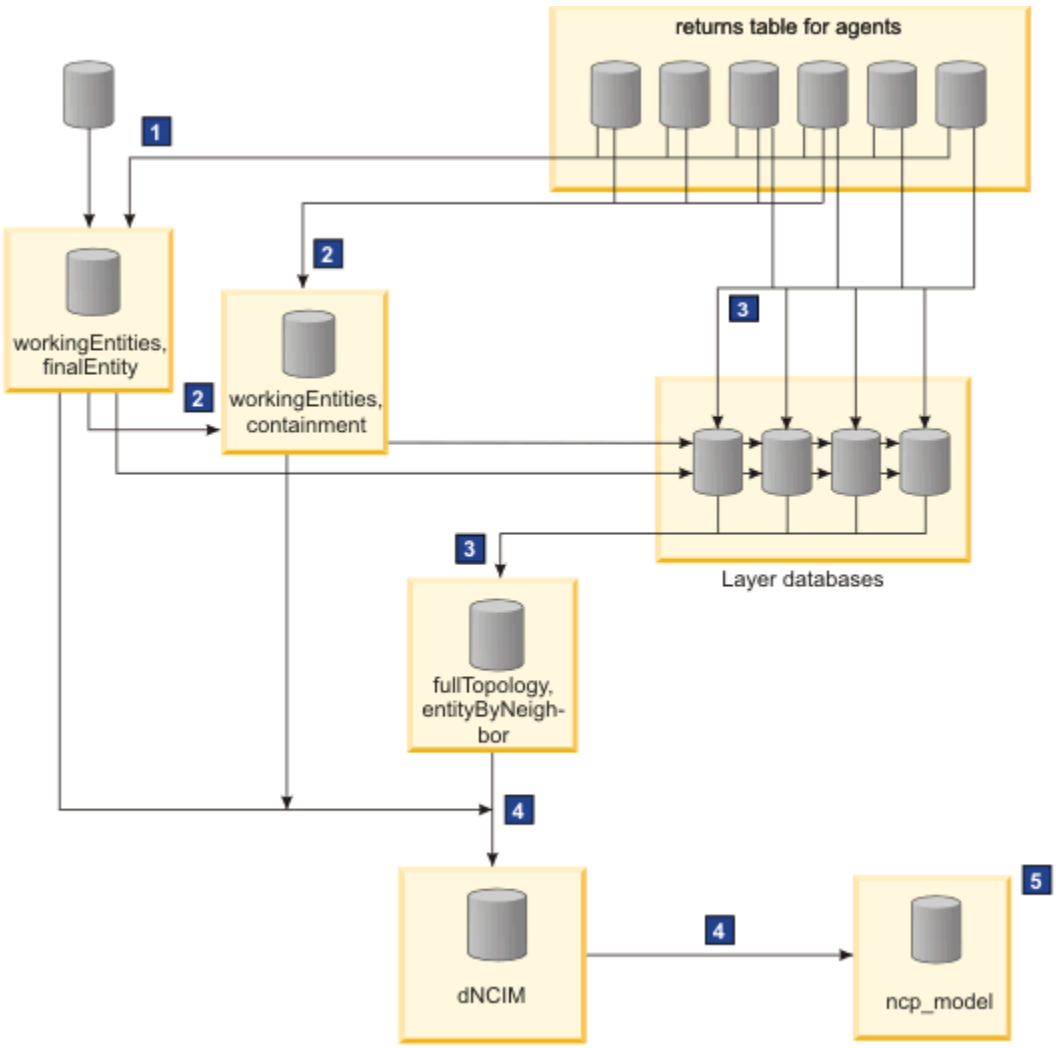


Figure 36. Discovery process flow: creating the topology

The following process flow describes the data flow.

1: After all the discovery agents have finished, and the discovery enters the data processing stage, special data processing stitchers interact with the discovery agent databases to produce the workingEntities.finalEntity table. Network entities that did not pass the instantiation filter are not included in the workingEntities.finalEntity table.

2: The stitchers use a subset of the agents-returns tables, together with the workingEntities.finalEntity table, to deduce and create the containment model. This model is stored in the workingEntities.containment table.

3: The stitchers use a further subset of the agents-returns tables, together with the workingEntities.finalEntity table and the workingEntities.containment table, to build the various topology layers, which are stored in the layer database tables. The full set of layers is merged in the fullTopology.entityByNeighbor table.

4: The stitchers merge the three tables produced (workingEntities.finalEntity; workingEntities.containment; fullTopology.entityByNeighbor) to build the network model. The network model is stored in the dNCIM database. The dNCIM data processing stitchers are used to populate the dNCIM database. The database topology is then sent in ncimCache format to ncp_model.

5: The Topology manager, `ncp_model`, receives the topology from the dNCIM database and merges it into the existing NCIM database topology model. The ncp_model process instantiates each network element and sends the topology to other components as required.

Advanced discovery configuration options

Use this information to understand how to configure the discovery process data flow and to configure download of full routing tables.

Configurable discovery data flow

The discovery process data flow is user-configurable. Stitchers control the movement of data between databases, and you can customize the discovery process by changing the way in which the stitchers are triggered and operate.

Stitcher and agent triggers

You can modify the data flow by changing the criteria that trigger the deployment of the stitchers and discovery agents, by modifying the stitchers, and, if necessary, by modifying the agent definitions. Some typical triggers are:

- Data being inserted into a specific database table
- A stitcher or discovery agent completing its operation
- The end of a discovery phase

Any changes you make are automatically detected by DISCO during its periodic scan of the agent and stitcher files (the scan frequency is determined by the entry in the disco.config database). On detecting changes, DISCO modifies its agent and stitcher definitions databases accordingly, and applies the changes to the next discovery cycle.

For more details about the stitchers and the stitcher language, see the *IBM Tivoli Network Manager Reference*.

On-demand stitchers

Stitchers can be started on demand. If you insert a stitcher into the stitchers.actions database, DISCO automatically runs the stitcher. This means that the discovery cycle can be started at any point, and further actions can be configured to start when the stitcher completes.

Partial matching

By default, the discovery process uses partial matching, which means that the discovery agents do not need to download the full routing tables during discovery.

You do not need to modify the discovery agent definition files to use partial matching. However, it is possible to prevent the IpForwardingTable and IpRoutingTable discovery agents from using partial matching in certain cases if you have devices on your network that do not support partial matching.

To prevent partial matching on certain devices, you must specify the devices that do not support partial matching in the `DiscoRouterPartialMatchRestrictions();` section of the `IpForwardingTable.agnt` definition file (for modern devices that use RFC2096) or the `IpRoutingTable.agnt` definition file (for older devices that use RFC1213). If a discovered device matches the filter specified in the `DiscoRouterPartialMatchRestrictions();` section, partial matching is not attempted on that device.

Discovery process with EMS integration

Network Manager collects topology data from an EMS using collectors.

The following steps show how Network Manager collects topology data from an EMS using collectors.

Collector-based discovery can be divided into the following conceptual steps:

- Discovering device existence
- Discovering basic device information

- Discovering detailed device information

For an overview of how Network Manager collects topology data from Element Management Systems (EMSs) and integrates this data into the discovered topology, see the *IBM Tivoli Network Manager User Guide*.

Discovering device existence with collectors

During a collector discovery, the discovery of device existence takes place in several steps.

Figure 37 on page 808 shows how the initial existence of devices held on the collectors is discovered.

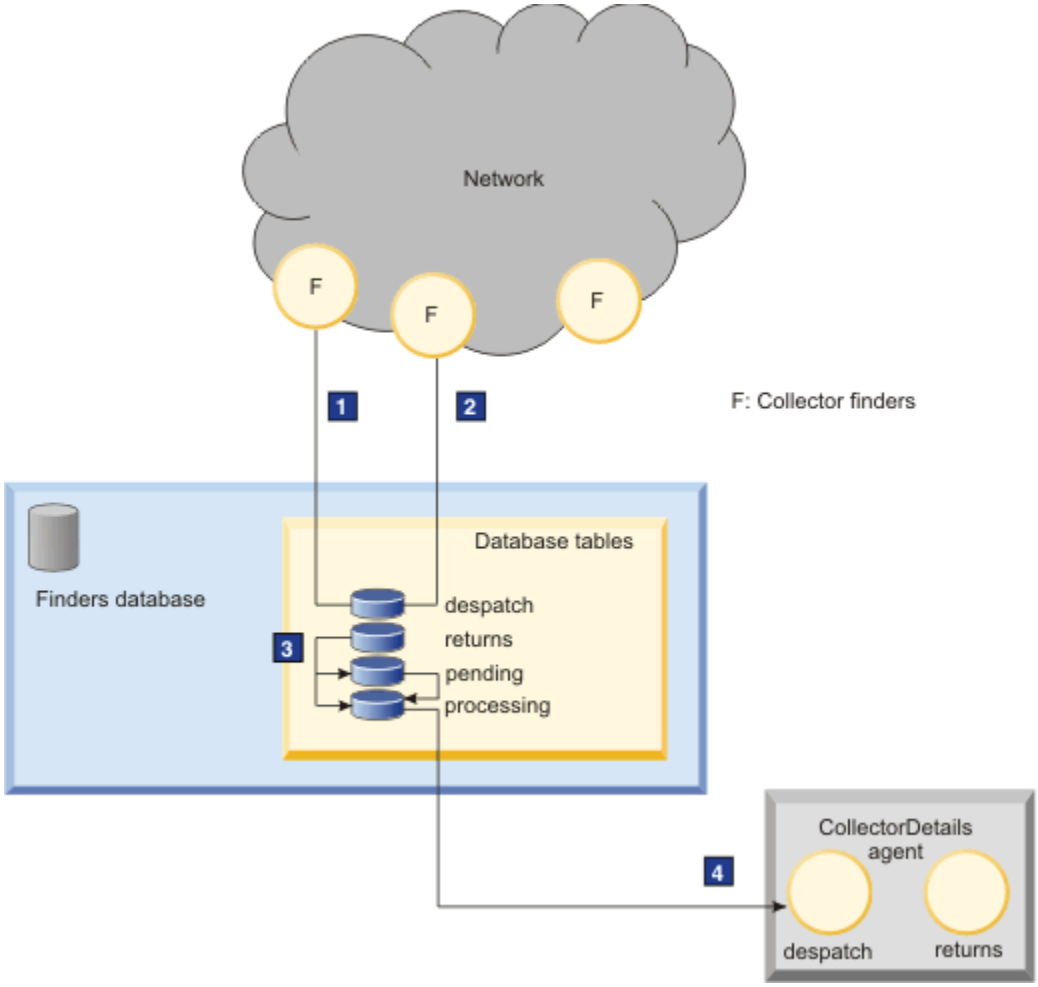


Figure 37. Collector discovery process flow: discovery of device existence

The following process flow describes Figure 37 on page 808:

1: The collector finders receive instructions from its configuration files and then proceeds to the network to look for collectors.

2: The collector finders return the list of devices to the finders.returns table.

3: Immediately after the device existence information is placed into the finders.returns table, the FnderRetProcessing stitcher moves the information to the finders.processing table, to denote that the network entity is being processed. If the discovery is in the blackout state, the information is placed into the finders.pending table.

4: The FnderProcToDetailsDesp stitcher moves the information about device existence from the finders.processing table to the CollectorDetails.despatch table so that the CollectorDetails agent can process the information.

Discovering basic device information

During a collector discovery, the discovery of basic device information takes place in several steps.

The following figure shows how basic device details are discovered in a collector discovery.

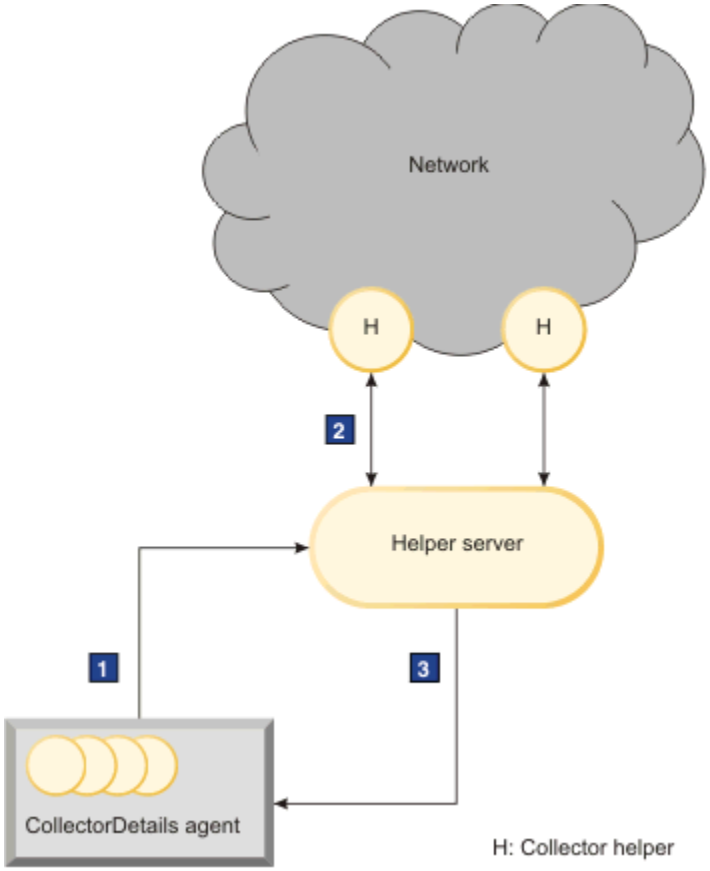


Figure 38. Collector discovery process flow: Discovery of basic device information

The following process flow describes Figure 38 on page 809:

1: All the agent despatch tables are active, so an insertion into the CollectorDetails.despatch table automatically triggers the CollectorDetails agent to discover basic device information from the collector.
2: The CollectorDetails agent uses the Helper Server to interrogate the helper collector .
3: The information retrieved from the network is returned to the CollectorDetails.returns table.

Discovering detailed device information

During a collector discovery, the discovery of detailed device information takes place in several steps.

The following figure shows how detailed device information is discovered in a collector discovery.

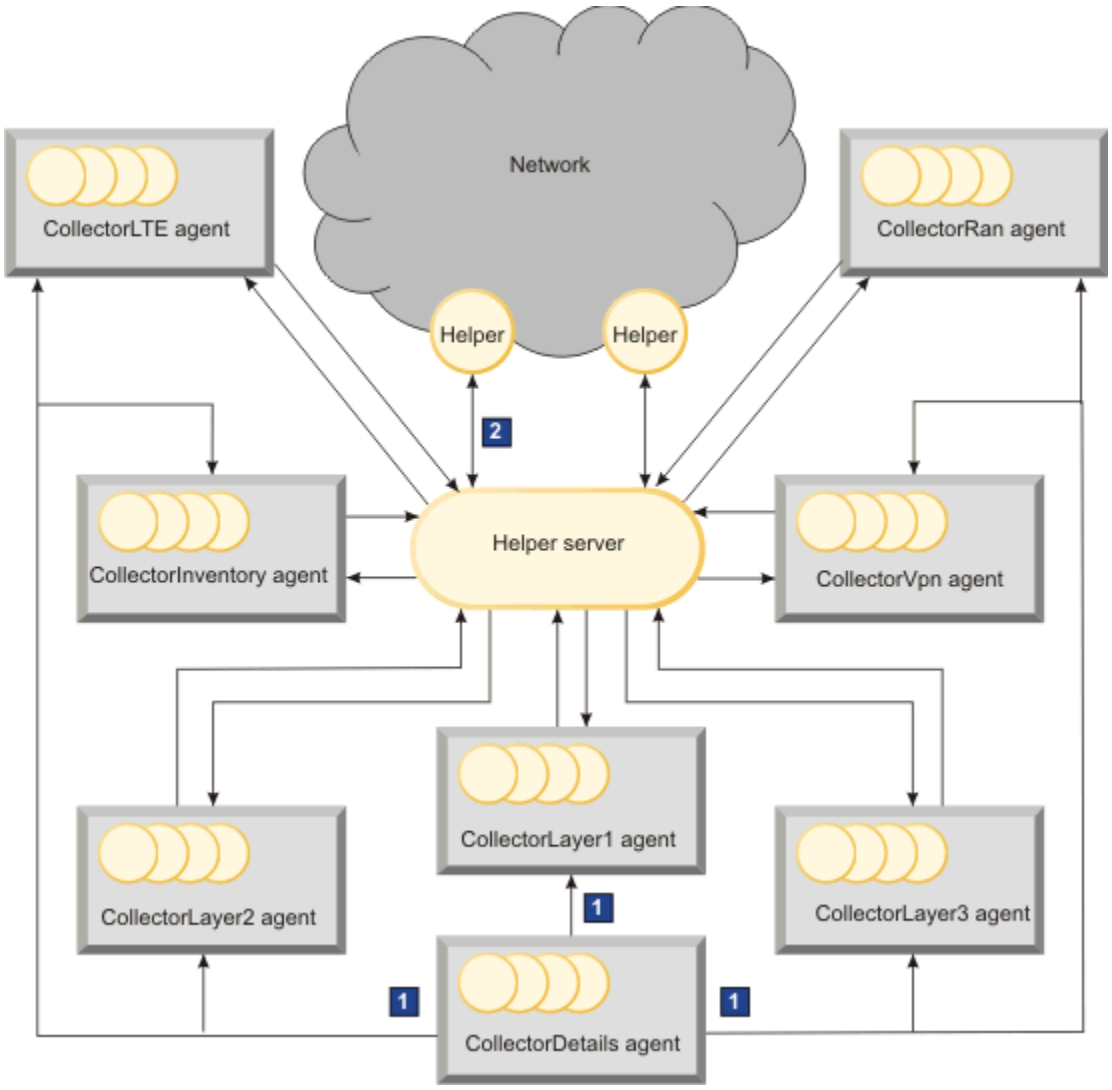


Figure 39. Collector discovery process flow: detailed device information

The following process flow describes Figure 39 on page 810:

1: The CollectorDetailsRetProcessing stitcher passes the information in the CollectorDetails.returns table to the despatch table of the various collector agents for processing.

2: Inserting information into the despatch table of an agent triggers an attempt by that agent to discover information about that device. The collector agents interrogate the collectors to discover the following information about each device.

CollectorInventory agent

Retrieves local neighbor, entity and associated address data for each of the devices on the collector.

CollectorLayer1

Retrieves layer 1 and microwave connectivity information for the devices on the collector.

CollectorLayer2

Retrieves layer 2 connectivity information for the devices on the collector.

CollectorLayer3

Retrieves layer 3 connectivity information for the devices on the collector.

CollectorLTE

Retrieves LTE-specific entity information for the devices on the collector.

CollectorRan

Retrieves radio access network (RAN) information for the devices on the collector.

CollectorVpn

Retrieves layer 2 and layer 3 VPN data for the devices on the collector.

Rediscovery

When a discovery has completed, **ncp_disco** enters rediscovery mode, in which the discovery of new devices results in updates to the topology model.

Full and partial rediscovery

By modifying the stitchers, you can configure the way DISCO treats devices that are found in the rediscovery mode.

By default, when the system is in rediscovery mode and either a new device is found or an existing device changes, the device is rediscovered. The stitchers ensure that the device is rediscovered only once. The stitchers also check that the change has not caused the relationship of the device with its neighbors to change. If necessary, the neighbors of the device are rediscovered. If the number of devices that need to be rediscovered as a result of relationship changes exceeds a certain limit, the rediscovery process initiates a full rediscovery.

Process flow of the FnderRetProcessing stitcher

To configure the way in which DISCO handles newly discovered devices, edit the FnderRetProcessing.stch stitcher. This stitcher processes the entries that are placed into the finders.returns table.

The default process flow of the FnderRetProcessing.stch stitcher is:

1. When an entry is placed in the finders.returns table, the stitcher checks whether the device is in the scope of the discovery. If the device is not in scope, it is ignored.
2. If the device is in scope and disco.status.m_DiscoveryMode=0, that is, DISCO is in discovery mode, the stitcher moves the device details to either the finders.pending table to be processed later (if the discovery is in the blackout state) or the finders.processing table to be processed now.
3. If the device is in scope and disco.status.m_DiscoveryMode=1, that is, DISCO is in rediscovery mode, the stitcher determines whether the device needs to be rediscovered. By default, the stitcher rediscovers:
   - Devices for which finders.returns.m_Creator='Rediscovery'. There is no Rediscovery finder, but this column is set to 'Rediscovery' by other stitchers, such as ProcRemoteConns.stch, to indicate that as a result of rediscovering other devices this device needs to be rediscovered.
   - Any newly found device that is in scope and has not already been discovered.
4. If necessary, you can alter the section of the FnderRetProcessing.stch stitcher that performs the above checks in order to configure when rediscovery of a device takes place, although this configuration adjustment must be undertaken by advanced users only.
5. If a device that has already been discovered is to be rediscovered, the stitcher refreshes the information held in the Helper Server that relates to that device.
6. For all devices to be rediscovered, the stitcher removes the old entries for that device from the finders.processing, Details.returns and Details.despatch tables, copying the old information to the rediscoveryStore.dataLibrary table for comparison purposes.
7. The stitcher then places the details of the device to be rediscovered into the finders.processing table and the FnderProcToDetailsDesp.stch stitcher moves the device details to the Details agent.

Processing information from discovery agents during rediscovery

After the entity that is being rediscovered has been processed by the Details agent, and the details are placed in the Details.returns table, the DetailsRetProcessing.stch stitcher compares the old data in the rediscoveryStore.dataLibrary table with the new data. By default, the rediscovery continues from this point.

If necessary, you can edit the DetailsRetProcessing.stch stitcher so that rediscovery continues only when certain conditions are in place. For example, rediscovery continues only when SNMP access is available.

The rediscovery data is processed by the AssocAddress agent and then by the appropriate agents (according to the configured discovery process flow) and sent to their returns tables.

A full discovery combines the information from the discovery agent returns tables to produce the topology. However, in a rediscovery, the information must be checked to determine whether the relationships between devices have changed as a result of the new information.

For example, if the device being rediscovered, device A, was connected to device B before the rediscovery, but is now connected to a third device, device C, then both B and C must also be rediscovered because their relationship has changed. The AgentRetProcessing.stch stitcher determines the relationships between devices and the comparison is done by ProcRemoteConns.stch. Switches and hubs need to be rediscovered differently to routers because the connectivity information they provide is indirect instead of direct. Any entity that also needs to be rediscovered as a consequence of rediscovery is inserted back into the finders.returns table with the parameter m_Creator='Rediscovery'.

Full rediscoveries

Comparing the current relationship between devices to their previous relationship, and rediscovering all the devices whose relationships have changed, can sometimes become circular. However, the discovery process includes a check to prevent this repetition.

If the ratio of entities that have been compared to the entities that need to be rediscovered exceeds the percentage specified in the disco.config.m_PendingPerCent column, then DISCO stops rediscovering individual devices and initiates a full network discovery.

Additionally, the fact that all rediscovered entities are recorded in the rediscoveryStore.rediscoveredEntities table means that a given entity can be rediscovered only once.

Rediscovery completion

When all the entities that need to be rediscovered have been processed, the topology layers are rebuilt by the FinalPhase.stch stitcher. This stitcher also clears the rediscoveryStore database so that it is ready for the next rediscovery.

It is important to note that DISCO might go through many discovery cycles during rediscovery before the topology is rebuilt. DISCO rebuilds the topology **only** when there are no entities needing to be rediscovered.

Option to rebuild topology layers

You can specify whether to rebuild the topology layers following a partial rediscovery. Using this option, you can increase the speed of partial rediscovery.

Suggested reasons to rebuild or not to rebuild the topology layers are:

- If you specify that the topology layers should *not* be rebuilt following partial rediscovery, the result is that new devices are added to the topology much faster than they would be if the topology layers are rebuilt; however, the resulting topology may not be complete. Connectivity associated with the newly discovered device is not fully reflected in the topology. Topology layers are fully rebuilt when a full rediscovery is run.
- If you specify that topology layers *should* be rebuilt following partial rediscovery, the result is an accurate topology showing all connectivity. However the process of adding new devices takes longer.

Use the m_RebuildLayers field in the disco.config table to specify whether or not to rebuild topology layers following partial rediscovery. You set this value as follows:

- If disco.config.m_RebuildLayers=0, then following partial rediscovery, topology layers stitchers are not run. The result is a much quicker partial discovery: however, connectivity associated with the newly discovered device is not fully reflected in the topology.
- If disco.config.m_RebuildLayers=1, then following partial rediscovery, topology layers stitchers are run. Partial rediscovery takes longer but results in a complete topology.

Chapter 26. Discovery agents

Use this information to support the selection of discovery agents to run as part of your discovery.

The following topics provide information on the discovery agents available. There is also guidance on the agents to select, based on the characteristics of your network.

Agents

Discovery agents retrieve information about devices in the network. They also report on new devices by finding new connections when investigating device connectivity. Discovery agents are used for specialized tasks. For example, the ARP Cache discovery agent populates the Helper Server database with IP address-to-MAC address mappings.

In addition to the main discovery agents, which can be enabled or disabled according to your discovery requirements, there are two agents that must always be run: the Details agent and the Associated Address agent.

Each discovery agent has its own database resident within DISCO. These databases are generically structured and based on a template called the agentTemplate database.

Each discovery agent database contains the following tables:

- *agentName*`.despatch`
- *agentName*`.returns`

Note: The default configuration sets the majority of agents to run. This is because the more agents that are run, the wider the range of networks that can be discovered. Furthermore, agents are designed to quickly stop analyzing devices that do not provide the data they require. This means that running a large number of agents increases network traffic by a very small amount only.

Note: Network Manager kills all discovery agents at the end of data collection stage 3. This ensures that the next discovery restarts the agents and forces the agents to reread their configuration files at the beginning of a discovery, thereby detecting any changes to the configuration files.

Details agent

This agent is triggered by the entries in the `finders.processing` table. At least one finder is needed to activate this agent. The SNMP helper configuration for associated devices is also a prerequisite for running this agent.

The Details agent retrieves basic information about devices discovered by the finders, and determines whether SNMP access to the device is available. This mandatory agent is triggered by the entries in the `finders.processing` table, so at least one finder is needed to activate this agent.

The Details agent is triggered when device information (usually transferred from the finders by a stitcher) is placed in the `Details.despatch` database table.

The Details agent retrieves basic information from the network and deposits this information in the `Details.returns` table. The basic information retrieved comprises the DNS name of the device obtained by the configured DNS helper, and the system object ID obtained by the SNMP helper. `IpForwarding` data is downloaded and inserted into the `ExtraInfo` field which is used to identify routing devices. `SysName` information is also downloaded for use if this optional naming scheme is required. The insertion of data into the `returns` table triggers a stitcher that sends this information to the Associated Address agent.

Associated Address (AssocAddress) agent

This mandatory agent is triggered by the output of the Details agent. The SNMP helper configuration for associated devices is a prerequisite for running this agent.

When an interface on a device has been discovered, and basic device information has been retrieved by the Details agent, a stitcher passes the discovered device information to the Associated Address agent. This agent downloads all the other IP addresses associated with the device and adds them to a central registry, held in the `translations.ipToBaseName` table, provided the device details are not already in the registry. Downloading all the associated IP addresses ensures that any given device is only interrogated once by the main discovery agents, thus reducing the load on the agents. Any attempt to discover a device more than once (using multiple interfaces) is blocked by the Associated Address agent as the device details are already in the `translations` database.

Provided the device being checked has not already been discovered, a stitcher sends the device details to the appropriate discovery agent for the retrieval of device connectivity and protocol-specific information.

Interface data retrieved by agents

The Interfaces agent downloads interface information primarily from the interfaces table of RFC1213.mib. For each device discovered, the interface information is written to the m_LocalNbr field within each record in the relevant agent.returns table.

The interface information can hold a number of subfields, including an index number that identifies the interface together with the properties of that interface and values for each property. For example, the m_LocalNbr field may include the following subfields:

- m_LocalNbr->m_IfIndex: the index associated with this interface
- m_LocalNbr->m_IfType: the type of interface
- m_LocalNbr->m_SubnetMask: the subnet mask of the interface

Discovery agent definition file keywords

Discovery agent definition file keywords are used to define the operation of discovery agents.

DiscoAgentClass

The `DiscoAgentClass` keyword specifies the basic type of agent. The following table identifies the most commonly used values:

Value	Description
0	Specifies an IP type agent.
1	Specifies a switch type agent.
2	Specifies a hub type agent.
3	Specifies an ATM device type agent.
4	Specifies an FDDI type agent.
5	Specifies a PVC type agent.
6	Specifies a frame relay type agent.
8	Specifies a NAT gateway agent.

The following example shows a `DiscoAgentClass` keyword set to a frame relay type agent. Frame relay type agents typically discover Frame Relay interfaces and connections between two points on Frame Relay networks that incorporate specific network devices, for example, CISCO devices.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentClass( 6 );
.
.
.
}
```

DiscoAgentClassEnabledByDefault

The `DiscoAgentClassEnabledByDefault` keyword specifies whether the agent is enabled by default for full discoveries. Specify one of the following values:

Value	Description
0	Specifies that the agent is not enabled by default for full discoveries.
1	Specifies that the agent is enabled by default for full discoveries.

The following example shows a `DiscoAgentClassEnabledByDefault` keyword set to enable a frame relay type agent by default for full discoveries.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentClass( 6 );
.
.
.
DiscoAgentEnabledByDefault( 1 );
}
```

DiscoAgentClassEnabledByDefaultOnPartial

The `DiscoAgentClassEnabledByDefaultOnPartial` keyword specifies whether the agent is enabled by default for partial discoveries. Specify one of the following values:

Value	Description
0	Specifies that the agent is not enabled by default for partial discoveries.
1	Specifies that the agent is enabled by default for partial discoveries.

The following example shows a `DiscoAgentClassEnabledByDefaultOnPartial` keyword set to enable a frame relay type agent by default for partial discoveries.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentClass( 6 );
.
.
.
DiscoAgentEnabledByDefaultOnPartial( 1 );
DiscoAgentEnabledByDefault( 1 );
}
```

DiscoAgentIsIndirect

A direct agent returns relationship data about objects that it is directly connected to at the layer it deals with. An indirect agent returns relationship data about objects it is indirectly connected to. The most common indirect agents are switch agents. The remote neighbor records for indirect agents relate to devices that can be reached from a specific port, not from devices to which they are directly connected. The relationship data from indirect agents is required to determine which remote neighbor records of a device need to be rediscovered when the device changes.

The `DiscoAgentIsIndirect` keyword specifies whether the agent is an indirect agent that returns relationship data about objects it is indirectly connected to. Specify one of the following values:

Value	Description
0	Specifies that the agent is a direct agent.
1	Specifies that the agent is an indirect agent.

The following example shows a `DiscoAgentIsIndirect` keyword set to specify that a frame relay type agent is a direct agent.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentGUILocked( 0 );
DiscoAgentClass( 6 );
DiscoAgentIsIndirect( 0 );
.
.
.
DiscoAgentEnabledByDefaultOnPartial( 1 );
DiscoAgentEnabledByDefault( 1 );
}
```

DiscoAgentCompanionAgents

The `DiscoAgentCompanionAgents` keyword is used to display in the GUI the agent or agents that this agent should execute with.

The following example shows a `DiscoAgentCompanionAgents` keyword that displays in the GUI the agent (`ArpCache.agnt`) that should execute with the Centillion Networks agent.

```
DiscoCompiledAgent
{
.
.
.
-- This agent examines all devices originally made by Centillion
-- Networks (enterprise OID 1.3.6.1.4.1.930), to see if it can
-- discover them.
.
.
.

DiscoAgentCompanionAgents( "ArpCache" );
.
.
.
}
```

DiscoAgentCompletionPhase

The `DiscoAgentCompletionPhase` keyword specifies during which of the discovery phases the specified agent should complete executing. Specify one of the following values:

Value	Description
1	Specifies that the agent should complete execution during discovery phase 1.
2	Specifies that the agent should complete execution during discovery phase 2.
3	Specifies that the agent should complete execution during discovery phase 3.

The following example shows a `DiscoAgentCompletionPhase` keyword set to enable a frame relay type agent to complete execution during discovery phase 1.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentCompletionPhase( 1 );
.
.
.
DiscoAgentEnabledByDefaultOnPartial( 1 );
DiscoAgentEnabledByDefault( 1 );
}
```

DiscoAgentConflictingAgents

The `DiscoAgentConflictingAgents` keyword is used to display in the GUI the agent or agents that this agent should not execute with.

The following example shows a `DiscoAgentConflictingAgents` keyword that displays in the GUI the agents (`IpRoutingTable.agnt` and `IpForwardingTable.agnt`) that should not execute with the IP backup routes agent.

```
DiscoCompiledAgent
{
.
.
.
-- This agent examines every device with SNMP access to see if it
-- can discover it.
.
.

DiscoAgentConflictingAgents( "IpRoutingTable","IpForwardingTable" );
.
.
.
}
```

DiscoAgentDescription

The `DiscoAgentDescription` keyword specifies a description of the agent that is displayed in the GUI.

The following example shows a `DiscoAgentDescription` keyword that specifies a description to display in the GUI for a frame relay type agent. The description makes use of HTML coding.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentDescription("
<b>Agent Name :</b>  CiscoFrameRelay<br>
<br>
<b>Agent Type :</b> Layer 3<br>
<br>
<b>Agent Prerequisites :</b>  SNMP helper configuration for associated devices.<br>
<br>
<b>Operation :</b><br>
Discovers Frame Relay interfaces and connections between two points on Frame Relay
networks that incorporate Cisco devices. If you need to add DLCI information to the
```

```
interfaces of Frame Relay devices, then run Frame Relay agents in conjunction with
the IP layer agents.<br>
<br>
    ");
.
.
.
}
```

DiscoAgentMinCertifiedDeviceOS

The DiscoAgentMinCertifiedDeviceOS keyword specifies a device operating system specific filter. This filter can be configured to run the specified agent against specific releases of a device operating system.

The following example shows a DiscoAgentMinCertifiedDeviceOS keyword that specifies a device operating system specific filter for an agent that discovers MPLS VRFs, VPNs, and label switching information from CISCO routers. This device operating specific filter configures the agent to run against the following CISCO devices and associated operating system releases:

- m_ObjectId — Specifies the CISCO devices (OID 1.3.6.1.4.1.9) that the agent attempts to discover.
- m_OSVersion — Specifies the CISCO device operating system filter that configures the agent to run against the following device operating system versions:
  - 12.0 releases of 12.0(27) or later which are not experimental
  - 12.2 releases of 12.2(19) or later which are not experimental
  - 12.3 releases of 12.3(18) or later which are not experimental
  - 12.4 releases

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentMinCertifiedDeviceOS
    (
      "(
          m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.',
          m_OSVersion >= '12.0(27)' AND m_OSVersion < '12.1' AND m_OSVersion
                         NOT LIKE '.*Experimental.*',
          m_MibVar = 'sysDescr.0'
        ),
        (
          m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.',
          m_OSVersion >= '12.2(19)' AND m_OSVersion < '12.3' AND m_OSVersion
                         NOT LIKE '.*Experimental.*',
          m_MibVar = 'sysDescr.0'
        ),
      (
          m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.',
          m_OSVersion >= '12.3(18)' AND m_OSVersion < '12.4' AND m_OSVersion
                         NOT LIKE '.*Experimental.*',
          m_MibVar = 'sysDescr.0'
       ),
      (
          m_ObjectId LIKE '1\.3\.6\.1\.4\.1\.9\.',
          m_OSVersion >= '12.4',
          m_MibVar = 'sysDescr.0'
        )"
    );
.
.
.
}
```

DiscoAgentPrecedence

The `DiscoAgentPrecedence` keyword specifies which agent gets precedence when there is conflicting data from two agents. Specify a value that is greater than or equal to 0 (zero). The recommended range of values is from 1 to 100, where the higher the number the higher the precedence. The higher the precedence the more that agent data is correct. For example, if given conflicting data from a precedence 2 agent and a precedence 3 agent then the precedence 3 agent data is used.

The following example shows a `DiscoAgentPrecedence` keyword for a frame relay type agent set to a precedence of 2.

```
DiscoCompiledAgent
{
.
.
.
DiscoAgentGUILocked( 0 );
DiscoAgentClass( 6 );
DiscoAgentIsIndirect( 0 );
DiscoAgentPrecedence( 2 );
.
.
.
DiscoAgentEnabledByDefaultOnPartial( 1 );
DiscoAgentEnabledByDefault( 1 );
}
```

DiscoPerlAgent

The `DiscoPerlAgent` keyword specifies whether this `.agnt` file refers to a Perl agent.

The following example shows a `DiscoPerlAgent` keyword specified for a Perl based agent that extracts information about the operating system running on the device.

```
DiscoPerlAgent
{
.
.
.
DiscoAgentGUILocked( 0 );
    DiscoAgentClass( 0 );
    DiscoAgentIsIndirect( 0 );
    DiscoAgentPrecedence( 2 );
    DiscoAgentEnabledByDefaultOnPartial( 0 );
    DiscoAgentEnabledByDefault( 0 );
}
```

Types of agents

The agents supplied with Network Manager can be divided into categories according to the type of data they retrieve or the technology they discover.

Discovery agents that discover connectivity among Ethernet switches

Discovery agents that discover connectivity information between Ethernet switches have three main operational stages: gain access to the switch and download all the switch interfaces; discover VLAN information for the switch; download the forward database table for the switch.

A list of the discovery agents that handle Ethernet switches is shown in Table 522 on page 822.

Note: Before you enable these layer 2 agents, it is necessary to configure SNMP access. Some agents also require Telnet access and Telnet Helper configuration.

Table 522. Ethernet switch discovery agents	
Agent name	**Function**
AccelarSwitch	The AccelarSwitch agent contains the specialized methods for retrieving connectivity information from Accelar routing switches. These devices are now branded as the Nortel Passport 86xx series. This agent also discovers BayStack 450 and BayStack 470 devices. This agent downloads the switch forwarding database (FDB) table and the VLAN information for the device. The switch stitcher uses this information to resolve layer 2 Ethernet connectivity.
BayEthernetHub	The BayEthernetHub agent discovers hub cards that are manufactured by Bay. Connectivity information is downloaded from the hub and the connectivity is resolved by the HubFdbToConnections stitcher. Before you enable this agent, it is also necessary to configure the SNMP Helper.
CentillionSwitch	The CentillionSwitch agent contains the methods that are needed to retrieve and resolve information from the Centillion switching devices, in particular the enterprise-specific VLAN information.
ChipcomDistributedMM	The ChipcomDistributedMM agent discovers the Ethernet switch connectivity for 3Com CoreBuilder 5000 devices that contain distributed management modules.
ChipcomEthernetMM	The ChipcomEthernetMM agent is appropriate for Chipcom online concentrators that contain Ethernet Management Modules (EMMs), and discovers the Ethernet connectivity of Chipcom EMMs.
CiscoSRP	The CiscoSRP agent discovers the connectivity of networks by using the Spatial Reuse Protocol (SRP), that is, DPT Ring topologies. SRP is a layer 2 protocol that was developed by Cisco. It uses ‘side' information to identify adjacent neighbors in its ring topology. The CiscoSRP agent discovers connectivity of any device that supports the CISCO-SRP-MIB. The agent definition file is configured by default to accept only Cisco devices with any IOS version, except devices that are supported by the CiscoSRPTelnet agent. The agent accepts only devices that support the srpMacAddress MIB variable. IOS version 12.2(14)S7 and 12.2(18)S, used with NPE-G1 cards, are known to corrupt SNMP data. IOS version 12.2(15)BC1 is known to lack CISCO-SRP_MIB support.

Table 522. Ethernet switch discovery agents (continued)	
Agent name	**Function**
CiscoSRPTelnet	The CiscoSRPTelnet agent discovers the connectivity of networks by using the Spatial Reuse Protocol (SRP), that is, DPT Ring topologies. SRP is a layer 2 protocol that was developed by Cisco that uses ‘side' information to identify adjacent neighbors in its ring topology. The CiscoSRPTelnet agent discovers the connectivity of any device that supports the `show controllers srp` command. The agent definition file is configured to accept only Cisco devices that have an IOS that is known not to support the CISCO-SRP-MIB. Those IOS versions that have issues with SNMP discovery. IOS version 12.2(14)S7 and 12.2(18)S, used with NPE-G1 cards, are known to corrupt SNMP data. IOS version 12.2(15)BC1 is known to lack CISCO-SRP_MIB support. **Note:** Before you enable this agent, configure Telnet access and the Telnet Helper.
CiscoSwitchSnmp	The CiscoSwitchSnmp agent contains the specialized methods for retrieving information from Cisco switches by using SNMP. This agent uses different methods for finding VLANs and card or port to ifIndex mappings because different Cisco switches store this information in different MIB variables. The CiscoSwitchSnmp agent also discovers Virtual Port Channel (vPC) links from Forwarding Database (FDB) data.
CiscoSwitchTelnet	The CiscoSwitchTelnet agent contains specialized methods for retrieving connectivity information from Cisco switches by using Telnet. This agent uses different methods to find VLANs and card/port to ifIndex mappings because different Cisco switches store this information in different MIB variables. Only FDB tables are downloaded by using Telnet. All other information is downloaded by using SNMP. The Telnet commands that are used to obtain the FDB table are `show cam dynamic` and `show mac-address table`. Some devices might require enable mode to run the `show mac-address table` command. **Note:** Before you enable this agent, configure SNMP and Telnet access and the SNMP and Telnet Helpers.
CiscoVSS	The Cisco VSS agent discovers Virtual Switching System information from Cisco switches.
Corebuilder3ComSwitch	The Corebuilder3ComSwitch agent discovers links for the CoreBuilder 9000 layer 3 switches that are manufactured by 3Com.

Table 522. Ethernet switch discovery agents (continued)	
Agent name	**Function**
DasanSwitchTelnet	The DasanSwitchTelnet agent is responsible for the discovery of the layer 2 connectivity in the FDB/MAC table of Dasan switches. The agent was developed against the following devices: V5208 (OS 9.07)V5224 (OS 9.10). The agent is able to discover layer 2 connectivity, VLANs, and trunk ports. It is configured to run only against devices with a sysObjectID of 1.3.6.1.4.1.6296.* and that support the command `show vlan`. **Note:** Before you enable this agent, configure Telnet access and the Telnet Helper.
DefaultEthernetHub	This agent has a specialized class for dealing with semi-intelligent hubs.
EnterasysSwitch	The EnterasysSwitch agent provides layer 2 connectivity discovery by retrieving the FDB table and VLAN information from the device. The agent discovers layer 2 connectivity for devices that support the IEEE 802.1q or IEEE 802.1d standards, as modeled in the Q-BRIDGE-MIB and BRIDGE-MIB SNMP MIBs. **Note:** This agent is used for Enterasys devices that do not have SecureFast turned on.
ExtremeSwitch	The ExtremeSwitch agent obtains layer 2 connectivity information, EDP neighbors, and VLAN details from Extreme switches. The Extreme devices must be configured to enable SNMP access and dot1dFdbTable population to achieve a detailed layer 2 discovery. Send the following commands to each Extreme device: • `enable snmp access` • `enable dot1dFdbTable` This configuration change is only required on switches that are running a version of ExtremeWare® before 6.1.8.
Fix Pack 2 F5Switch	This agent discovers configuration for F5 switches. The agent retrieves information from the `sysChassisSlotSlotId` variable in the F5-BIGIP-COMMON-MIB and F5-BIGIP-SYSTEM-MIB MIBs. **Note:** Before you enable this agent, configure SNMP and Telnet access and the SNMP and Telnet Helpers.
FoundrySwitch	The FoundrySwitch agent discovers switch connectivity of any Foundry device that supports the IEEE 802.1q or IEEE 802.1d standards, as modeled in the Q-BRIDGE-MIB and BRIDGE-MIB SNMP MIBs. The agent definition file is configured to accept all SNMP-enabled Foundry devices by default. The agent discovers only devices that support the Q-BRIDGE-MIB dot1qVlanVersionNumber MIB variable or the BRIDGE-MIB. The FoundrySwitch agent also obtains multislot port trunking information, but does not discover single-slot port trunking. Some Foundry devices support only IEEE 802.1d and as a consequence no VLAN information is discovered for these devices.

Table 522. Ethernet switch discovery agents (continued)

Agent name	Function
HuaweiLLDPTelnet	The HuaweiLLDPTelnet agent discovers the Layer 2 physical switch-to-switch links between Huawei switches. To discover the Layer 2 connections between switch interfaces it uses the LLDP Telnet command `display lldp neighbor brief`. It does not use the SNMP method that retrieves the FDB (MAC) table and its entries on a switch, because this method does not show all the data from the LLDP MIB. This agent does not use the telnet command `display mac dynamic`, because this command has similar issues. Using this agent avoids getting duplicate MAC addresses on multiple ports or interfaces.
HuaweiSwitchTelnet	The HuaweiSwitchTelnet agent discovers the Ethernet switch connectivity for Huawei Quidway switches. This agent is Telnet-based, but also requires SNMP access to discover certain information. It requires completion of the Privileged mode (Super 3 mode) sections of the `TelnetStackPasswords.cfg` configuration file. If you do not complete these parts of the file, the agent fails. Certain Telnet commands have the side-effect of changing the command prompt of a Huawei device. For example, the command prompt: <device_name> becomes `[device_name]` or `[device_name-diag]` when certain commands are entered. The parameters m_ConPrompt and m_PrivConPrompt in the `TelnetStackPasswords.cfg` file must be configured to cope with these variations. **Note:** Before you enable this agent, configure Telnet access and the Telnet Helper.
HPSwitch	The HPSwitch agent contains the specialized methods for retrieving connectivity information from HP ProCurve switches, including the download of enterprise-specific VLAN information.
Marconi3810	The Marconi3810 specialized agent discovers the Ethernet connectivity of Marconi ES-3810 switches that run operating system version 4.x.x and 5.x.x. This agent also removes connectivity from LANE interfaces by default, which can be configured by using the GetElanData flag in the `.agnt` file.

Table 522. Ethernet switch discovery agents (continued)	
Agent name	**Function**
SecureFast	The SecureFast agent contains the specialized methods for retrieving connectivity information from Enterasys/Cabletron SecureFast VLAN switches. These devices use the Cabletron Discovery Protocol to discover their neighbors. The SecureFast operating mode is turned on. This agent is sent to all Cabletron and Enterasys devices that are specified by 1.3.6.1.4.1.52.* and 1.3.6.1.4.1.5624.* in the `.agnt` file, and determines whether a device is SecureFast enabled by downloading the sfpsCommonNeighborSwitchMAC MIB variable. Devices in SecureFast mode do not support the dot1dBridge MIBs.
StandardSwitch	The StandardSwitch generic agent provides layer 2 connectivity discovery of all switches for which a specialized agent does not exist. The agent discovers layer 2 connectivity for devices that support the IEEE 802.1q or IEEE 802.1d standards, as modeled in the Q-BRIDGE-MIB and BRIDGE-MIB SNMP MIBs. Devices in SecureFast mode do not support the dot1dBridge MIBs.
SuperStack3ComSwitch	The SuperStack3ComSwitch agent finds the connections in stacked switches that are manufactured by 3Com.
XyplexEthernetHub	The XyplexEthernetHub agent discovers the layer 2 connectivity of intelligent hubs that are manufactured by Xyplex.

Connectivity at the layer 3 network layer

There are a number of discovery agents that retrieve connectivity information from OSI model layer 3 (the *Network Layer*). Layer 3 is responsible for routing, congestion control, and sending messages between networks.

Table 523. Layer 3 network layer agents	
Agent name	**Function**
AlteonVRRP	VRRP is not modelled for RCA. The AlteonVRRP agent only sets tags on VRRP interfaces that show the state of Alteon routers at the time of the discovery. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
CiscoBGPTelnet	The CiscoBGPTelnet agent downloads the following BGP data from Cisco routers: • Peer data: the agent retrieves iBGP and eBGP data from peer routers. • Route data: the agent retrieves routing information from BGP routing tables of peer routers. This option is off by default as it will retrieve huge amounts of data from a typical service provider network. This agent also provides the option to configure a filter to specify the route data that you would like to retrieve. **Note:** Before enabling this agent, configure Telnet access and the Telnet helper.

Table 523. Layer 3 network layer agents (continued)	
Agent name	**Function**
CiscoFrameRelay	The CiscoFrameRelay agent discovers Frame Relay interfaces and connections between two points on Frame Relay networks that incorporate Cisco devices. Frame Relay agents should be run in conjunction with the IP layer agents if you want to add DLCI information to the interfaces of Frame Relay devices. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
CiscoOSPFTelnet	The CiscoOSPFTelnet agent is responsible for discovery of Cisco devices running the Open Shortest Path First (OSPF) protocol. This agent provides complementary information to that of the StandardOSPF agent, such as what OSFP processes are running and virtual-link information. **Note:** Before enabling this agent, configure Telnet access and the Telnet helper.
ExtremeESRP	The ExtremeESRP agent discovers Extreme Standby Routing Protocol (ESRP) information from Extreme routing switches. ESRP is a feature of ExtremeWare that allows multiple switches to provide redundant routing services to users. The agent relies on the `extremeEsrpTable` and `extremeEsrpNeighborTable` of the EXTREME-ESRP-MIB being correctly populated. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
FoundryVRRP	VRRP is not modelled for RCA. The FoundryVRRP agent only sets tags on VRRP interfaces that show the state of Foundry routers at the time of the discovery. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
HSRPSnmp	The HSRPSnmp agent retrieves connectivity information using SNMP by means of the MIB from routing devices that use the HSRP (Hot Stand-by Routing Protocol) Virtual IP protocol. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
InetRouting	The InetRouting agent discovers connectivity.
Interfaces	This agent is triggered by the AssocAddress agent returns. The Interfaces agent downloads interface information primarily from the interfaces table of RFC1213.mib. The information will then be written to the m_LocalNbr field of the returned entities. You can increase or decrease the number of returned variables by modifying the Interfaces.agnt. Any basic MIB variable (sysDescr, sysName, and so on) or MIB variable that is indexed by the ifIndex can be added to the OIDs to download in the .agnt file. The Interfaces agent also retrieves IPv6 interface information. You must enable the Interfaces agent if you want to use interface filtering. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.

Table 523. Layer 3 network layer agents (continued)	
Agent name	**Function**
IpBackupRoutes	The IpBackupRoutes agent finds links by looking through the IpNetToMedia MIB table, which gives the physical and IP address of devices connected to the router. This agent is not enabled by default because it retrieves a large amount of information that is not essential in order to determine layer 3 connections. Furthermore, this information may be obsolete because it is downloaded from a table that is not dynamic and requires manual refresh. If you are performing a layer 2 discovery, then the server connectivity that this agent discovers is often obsolete, as it may have been superseded by switch connectivity information. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
IpForwardingTable	The IpForwardingTable agent finds links in the more recent version of the routing tables, that is, the IP Forwarding table as specified in RFC 2096. It also exploits Open Shortest Path First (OSPF) information to enhance the discovery of Juniper devices. This agent downloads elements from the routing table based on discovery scoping. The default setting assumes that the SNMP agent for a particular device supports partial matching. If the device cannot partial match, this should be specified in the DiscoRouterPartialMatchRestrictions section of the .agnt file. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
IpRoutingTable	Retrieves generic connectivity information by looking through the router routing table, as specified in RFC1213. The agent downloads elements from the routing table based on discovery scoping. The default agent setting assumes that the SNMP agents for particular devices support partial matching. If a device cannot partial match, this should be specified in the DiscoRouterPartialMatchRestrictions section of the .agnt file. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
ISISExperimental	Discovers connectivity between routers that support the experimental ISIS MIBs. This agent should be used when some of your routers are configured with netmasks of 255.255.255.255, making them unsuitable for standard discovery. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
LinkStateAdvOSPF	Retrieves link state advertisements (LSAs) from OSPF routers. These LSAs are used by the CreateOSPFNetworkLSAPseudoNodes stitcher to create OSPF pseudonodes. Pseudonodes overcome the problem of full meshing when representing OSPF area in Topoviz Network Views and enables connections within OSPF areas to be visualized in a clear, uncluttered manner.
JuniperBGPTelnet	Downloads BGP information from Juniper routers. It is not enabled by default because it gathers a very specific piece of information only, that is, whether devices are route reflectors. **Note:** Before enabling this agent, configure Telnet access and the Telnet helper.

Table 523. Layer 3 network layer agents (continued)	
Agent name	**Function**
NetScreenInterface	The NetScreenInterface agent retrieves information about all configured interfaces in Juniper Netscreen devices. The agent retrieves information about logical interfaces and other interfaces, which is not available from the standard IF-MIB, and requires both the NETSCREEN-INTERFACE-MIB.mib and NS-VPN-MON.mib files. The agent also retrieves VPN tunnel and tunnel connectivity information that is configured in Juniper NetScreen devices. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
NetScreenIpRoutingTable	The NetScreenIpRoutingTable agent retrieves information on IP routing tables configured on Netscreen devices. The agent determines the interfaces and sub-interfaces from the interface index of the Netscreen device. This agent performs the same function as the IpRoutingTable agent, but for Netscreen devices only, in order to take account of sub-interfaces which would not be discovered correctly by the IpRoutingTable agent. The NetScreenIpRoutingTable agent uses the IP-FORWARD-MIB Standard MIB and the NETSCREEN-INTERFACE-MIB. **Note:** The IpRoutingTable agent does not process the Netscreen devices processed by the NetScreenIpRoutingTable agent.
NokiaVRRP	Downloads VRRP information from routers that support the Nokia interpretation of the VRRP MIB. The information retrieved includes the VRRP state, ID, primary IP and associated addresses. This information is retrieved from the following MIB variables: • vrrpOperState • vrrpOperMasterIpAddr • vrrpAssoIpAddrRowStatus **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
NortelPassport	The NortelPassport agent retrieves Layer 3 connectivity and containment information from Nortel Passport switches.
RFC2787VRRP	The RFC2787VRRP agent downloads Virtual Router Redundancy Protocol (VRRP) information from routers that run RFC2787-compliant VRRP and support the RFC2787 VRRP MIB. Some Nokia firewalls support this MIB. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper. VRRP is not modelled for RCA. This agent sets tags on VRRP interfaces that show the state of the interfaces at the time of the discovery. The agent also downloads associated IP addresses, which are used to build VRRP collections. **Tip:** There are two subtly different versions of the VRRP MIB. They contain the same names but with different OIDs. If this agent does not work, use the other version of the VRRP MIB.

Table 523. Layer 3 network layer agents (continued)	
Agent name	**Function**
StandardBgp	The StandardBgp agent is responsible for discovery of networks running the Border Gateway Protocol. It supports any device that complies with the standard RFC1657 (BGP4-MIB) MIB and discovers the following information: • Autonomous System IDs • BGP Peer connections to external peers (EBGP) • BGP Peer connections to internal peers (IBGP) • BGP acquired route data (not recommended) The agent definition file is configured to accept all SNMP enabled devices by default, but the agent will only accept devices that support the BGP44-MIB, bgpIdentifier MIB variable. The agent has the following additional configuration parameters in the DiscoAgentDiscoveryScoping section of its .agnt file: • GetPeerData – determines whether the agent should acquire BGP Peer data (activated by default). • GetRouteData – determines whether the agent should acquire BGP routes (deactivated by default). This may result in a large amount of data being discovered. The StandardBgp agent does not currently support peer groups, confederations, per VRF BGP processes, or route reflection. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper. It is also necessary to configure the Ping helper.
StandardOSPF	The StandardOSPF agent is responsible for the discovery of networks running the Open Shortest Path First (OSPF) protocol. It will support any device that complies with the standard RFC1850. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.
TraceRoute	The TraceRoute agent finds links by tracing the route taken by an ICMP ping packet with a predetermined life span. If you are using this agent, you should increase the value of m_Timeout in the DiscoPingHelperSchema.cfg configuration file, as traceroute functionality takes longer than standard ICMP. This agent is not enabled by default as it does not only operate on SNMP-enabled devices. Therefore, if this agent were switched on by default, it would trace the route to every device on the network. The result could be incomplete connectivity in a meshed environment or inaccurate connectivity in a load-balanced environment. **Note:** Before enabling this agent, configure SNMP access and the SNMP helper.

Topology data stored in an EMS

There are several discovery agents that retrieve information about devices managed by an EMS.

The routing protocol discovery agents query an EMS collector for basic and detailed information about devices managed by EMS. These agents are shown in Table 524 on page 831.

Table 524. Routing protocol discovery agents	
Agent name	**Function**
CollectorDetails	Retrieves basic information about the devices on the collector, including sysObjectId, sysDescr, and naming data.
CollectorInventory	Retrieves local neighbor, entity and associated address data for each of the devices on the collector.
CollectorLayer1	Retrieves layer 1 and microwave connectivity information for the devices on the collector.
CollectorLayer2	Retrieves layer 2 connectivity information for the devices on the collector.
CollectorLayer3	Retrieves layer 3 connectivity information for the devices on the collector.
CollectorLTE	Retrieves LTE-specific entity information for the devices on the collector.
CollectorRan	Retrieves radio access network (RAN) information for the devices on the collector.
CollectorVpn	Retrieves layer 2 and layer 3 VPN data for the devices on the collector.

Discovering connectivity among ATM devices

Asynchronous Transfer Mode (ATM) is an alternative switching protocol for mixed format data (such as pure data, voice, and video). Several types of discovery agents can be used to discover ATM devices on a network.

Note: Before enabling these agents, it is necessary to configure SNMP access and the SNMP Helper.

Table 525. ATM discovery agents	
Agent name	**Function**
AtmForumPnni	The AtmForumPnni agent retrieves connectivity information from ATM devices that use the Private Network-to-Network Interface (PNNI) dynamic routing protocol and the ATM Forum's PNNI MIB. The PNNI protocol is commonly used on large networks, as it provides ATM switches with a detailed map of the network topology so that the ATM devices can make optimal routing decisions.
CellPath90	The CellPath90 agent enables discovery of the ATM connection of Marconi CellPath 90 WAN (Wide Area Network) multiplexers. The CellPath 90 WAN multiplexer does not know the ATM addresses of its neighbours, so it can only be discovered when it is connected to another, more intelligent, certified ATM device. The CellPath90 discovery agent is used in the calculation of network topology. It places information about the CellPath 90 into the correct layers within the discovery database.
CiscoPVC	The CiscoPVC agent retrieves PVC data from Cisco devices.
ILMI	The ILMI agent retrieves connectivity information from devices using the Interim Local Management Interface (ILMI), an RFC standard for managing ATM and IP networks. It investigates how ATM networks are connected down to the layer 2 virtual circuit and port level. This agent also removes logical connectivity from LANE interfaces.

Table 525. ATM discovery agents (continued)	
Agent name	**Function**
ILMIForeSys	The ILMIForeSys agent discovers physical ATM connections between devices by using the ILMI (Interim Local Management Interface) connectivity information provided by the Marconi ASX series of switches. When connectivity is deduced using ILMI information, it is usually the same as the connectivity that could have been calculated using PNNI information, as is the case with the standard AtmForumPnni and ILMI agents. However, there are some situations where the ILMI information contains details of a connection that is not in the PNNI information, and some situations where the PNNI information details a connection not in the ILMI information. The following examples detail situations where this may be the case: • Connections between ASX series switches and SE420/SE440 IADs are only discovered using ILMI. • Connections between Cisco routers or switches containing ATM cards and an ATM core are only discovered using ILMI. • As with the PnniForeSys agent, the ILMIForeSys agent is designed to operate seamlessly in conjunction with the ILMI agent. A network containing a mixture of ASX devices and another vendor's devices (for example, Cisco 5509 switches with ATM cards) can, therefore, be accurately discovered.
MariposaAtm	The MariposaAtm agent discovers the ATM connectivity of the SE420 and SE440 Integrated Access Devices (IADs). **Note:** The Ethernet switching and Frame Relay capabilities of these devices are not currently certified.
PnniForeSys	The PnniForeSys agent discovers physical ATM connections between devices by using the Private Network-to-Network Interface (PNNI) connectivity information provided by the Marconi ASX series switches. The PnniForeSys agent is designed to operate in conjunction with the AtmForumPnni agent. The PnniForeSys agent performs extra processing on Fore devices that do not store a logical ifIndex in their pnniLinkIfIndex variable. The information retrieved from these devices requires further processing to retrieve the actual ifIndex, which is held within the ifTable . **Note:** SNMP helper configuration for associated devices is a prerequisite for this agent. The AtmForumPnni agent must also be active.

Agents for discovering MPLS devices

To discover Multiprotocol Label Switching (MPLS) data, including Virtual Private LAN Service (VPLS) information, enable the appropriate agents.

The agents that retrieve MPLS data use either Telnet or SNMP to retrieve the data. Before you enable the MPLS agents, configure Telnet and SNMP access.

- Before you enable the MPLS agents that use Telnet, configure Telnet to enable the agents to access devices and to understand device output.
- Before you enable the MPLS agents that use SNMP, configure SNMP to enable access to devices and to specify threads, timeouts, and number of retries.

Tip: Agents that retrieve VPLS information can retrieve large amounts of data. Enabling these agents can add significant processing time to the discovery process. If you do not need to rediscover VPLS information, disable these agents for a faster discovery.

Table 526. MPLS discovery agents	
Agent name	**Function**
CiscoMPLSSnmp	The CiscoMPLSSnmp agent discovers MPLS paths on Cisco devices by using standard MIBs, and on Cisco devices that support the Cisco Experimental MPLS MIBs.
CiscoMPLSTelnet	The CiscoMPLSTelnet agent discovers MPLS paths and LDP VPLS on Cisco devices.
CiscoQinQTelnet	The CiscoQinQTelnet agent discovers QinQ (IEEE 802.1QinQ) configuration on Cisco devices.
HuaweiMPLSTelnet	The HuaweiMPLSTelnet agent discovers Layer 2 and Layer 3 MPLS/VPN related data on Huawei devices, including User-facing Provider Edge (UPE) and Network Provider Edge (NPE) information. You must configure the discovery to discover and visualize VPLS information from Huawei VPNs.
JuniperMPLSTelnet	The JuniperMPLSTelnet agent discovers MPLS paths on Juniper devices. This agent also discovers Juniper MultiHome VPLS configurations and tags the Virtual Switch Instance (VSI) with the relevant information.
JuniperMPLSSNMP	The JuniperMPLSSNMP agent discovers MPLS/VPN (RT-based VPN discovery) and VPLS (LDP and BGP) related data on Juniper devices.
JuniperQinQTelnet	The JuniperQinQTelnet agent discovers QinQ (IEEE 802.1QinQ) configuration on Juniper devices.
LaurelMPLSTelnet	The LaurelMPLSTelnet agent discovers MPLS paths on Laurel devices. This agent is intended for route target-based discoveries only.
StandardMPLSTE	The StandardMPLSTE discovers MPLS Traffic Engineered (TE) tunnels by using SNMP.
UnisphereMPLSTelnet	The UnisphereMPLSTelnet agent discovers MPLS paths on Juniper ERX routers (formerly Unisphere).

Multicast agents

Multicast agents retrieve data from devices participating in multicast groups and routes.

The agents that retrieve multicast data need SNMP and Ping access to retrieve the data. Before enabling the multicast agents, ensure that you first configured SNMP to enable the agents to access devices and to specify threads, timeouts, and number of retries.

The following table describes the multicast agents.

Table 527. Multicast discovery agents	
Agent name	**Function**
StandardIGMP	Discovers networks running the Internet Group Management Protocol (IGMP). Supports any device that complies with the RFC2933 IGMP MIB. Depending on the level of MIB support, the following information may be discovered: IGMP Interfaces; Per-Interface Group Memberships; Group Members Visible on IGMP Interfaces.

Table 527. Multicast discovery agents (continued)	
Agent name	**Function**
StandardIPMRoute	Discovers IP multicasting networks. Supports any device that complies with the RFC2932 IPMRoute MIB. Depending on the level of MIB support, the following information may be discovered: Multicast Routing data (upstream/downstream); Interfaces involved in Multicast Routing; Multicast Sources and Groups.
StandardPIM	Discovers networks running the Multicast protocol PIM. Supports any device that complies with the RFC2934 PIM MIB. Depending on the level of MIB support, the following information may be discovered: PIM Interfaces; PIM Adjacencies; Candidate RPs/BSR.

Discovering NAT gateways

There are several agents that download Network Address Translation (NAT) information from known NAT gateways.

None of the agents listed in the table below is enabled in the default configuration. These agents require advanced configuration, and it is preferable not to enable them by default.

Table 528. NAT gateway agents	
Agent name	**Function**
CiscoNATTelnet	The CiscoNATTelnet agent interrogates Cisco routers acting as NAT gateways. This agent downloads the static NAT translations by means of TELNET from the device. The translations are then used to identify within which part of the network a particular device exists. **Note:** Before enabling this agent, it is necessary to configure Telnet access and the Telnet Helper.
NATNetScreen	The NATNetScreen agent interrogates NetScreen® Firewalls acting as NAT gateways. This agent downloads the static NAT translations by means of TELNET from the device. The translations are then used to identify within which part of the network a particular device exists. **Note:** Before enabling this agent, it is necessary to configure Telnet access and the Telnet Helper.
NATTextFileAgent	The NATTextFileAgent mimics the function of the other NAT gateway agents by reading NAT mapping information from a flat file. The translations are then used to identify within which part of the network a particular device exists. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.

Discovering containment information

An important principle used by the network model is containment. A container holds other objects. You can put any object within a container and even mix different objects within the same container.

Containment information includes a physical breakdown of all parts held within the container, as well as detailed information on each of these parts. The parts that can be held within a container are:

- Chassis
- Interface
- Logical interface

- Vlan object
- Card
- PSU
- Logical collection, such as a VPN
- Module

There is also an Unknown category, which covers entities for which no part type has been defined.

The following table describes the discovery agents that discover containment information.

Table 529. Discovery agents that discover containment information

Agent name	Function
AvayaPhysicalInventory	The AvayaPhysicalInventory agent queries RAPID-CITY MIB for each physical entity and retrieves containment information for that physical entity. Run the AvayaPhysicalInventory agent if you want to model physical containment and perform asset management. Enable this agent if you have Avaya (formerly Nortel) devices in your network. **Note:** Configure SNMP access and the SNMP Helper before enabling this agent.
BrocadeEntity	The `BrocadeEntity` agent queries `FOUNDRY-SN-ROOT-MIB`, `IF-MIB` and `ENTITY-MIB` MIBs for Brocade ICX and VDX devices for each physical entity. The agent retrieves containment information for that physical entity. Run the `BrocadeEntity` agent if you want to model physical containment and perform asset management for `ICX6430`, `ICX7430`, `VDX8770` and `VDX6470` devices. Enable this agent if you have Brocade devices in your network. **Note:** Configure SNMP access and the SNMP Helper before enabling this agent.
IBMSystemNetworkingSwitch	The IBMSystemNetworkingSwitch agent retrieves Layer 2 connectivity and VLAN containment information (including VLAN tags, VLAN Trunk, and Trunk Group information) using SNMP.
Entity	The Entity agent queries the MIB for each entity and retrieves containment information for that entity. Before enabling this agent, you must configure SNMP access and the SNMP Helper. Running the Entity agent during a discovery is optional. Some containment information is gathered during a discovery even if the Entity agent is not run. Run the Entity agent to model physical containment and perform asset management. **Note:** During a discovery, the Entity agent retrieves a large amount of data. This slows down the discovery. You should therefore only use this agent if you need to perform asset management on the retrieved data. For information on Entity agent configuration, see "Entity agent configuration" on page 836.
IfStackTable	The IfStackTable determines the interface stacking hierarchy on devices that support the RFC 2863 MIB. **Note:** Configure SNMP access and the SNMP Helper before enabling this agent.
JuniperBoxAnatomy	The JuniperBoxAnatomy agent retrieves information about which modules and components are installed in a Juniper device and their containment. The agent uses vendor-specific MIBs such as the Juniper Box Anatomy MIB.

Table 529. Discovery agents that discover containment information (continued)	
Agent name	**Function**
JuniperERXIfStackTable	The JuniperERXIfStackTable determines the interface stacking hierarchy on Juniper ERX devices. This agent determines virtual-router and VRF context-sensitive stacking information for Juniper ERX devices. When a context-sensitive discovery is enabled this agent can be disabled, as the IfStackTable agent also determines this information. This will improve the performance of discovery. **Note:** Configure SNMP access and the SNMP Helper before enabling this agent.
JuniperLAGStack	The JuniperLAGStack agent retrieves Link Aggregation Group (LAG) information from Juniper devices. LAG information is needed to accurately represent the interface stacking hierarchy.
JuniperVlanTagTelnet	Discovers VLAN tagging configuration for Juniper E, ERX, M and MX Series routers. For Juniper model MX, M and T series, the agent issues the telnet command `show configuration interfaces` and captures the vlan-id from the output. For Juniper model E and ERX series, the agent retrieves juniVlanSubIfVlanId and juniVlanSubIfVlanStackId from the Juniper-ETHERNET-MIB. **Note:** Configure the SNMP Helper and the Telnet Helper before enabling this agent.

Entity agent configuration

You can configure the Entity agent to specify how much data the agent should retrieve. You can optionally choose to download this extra information from the entity MIBs of the Asset, ExtraPhysData, Module, Power, and Sensor entities.

Configure the Entity agent by setting the following variables in the `Entity.agnt` file:

- GetAssetData
- GetExtraPhysData
- GetModuleData
- GetPowerData
- GetSensorData

In each case, set a value of `1` to retrieve the data, and set a value of `0` if you do not want to retrieve the data. The default value is `1`.

In addition, you can specify how the Entity agent retrieves data from devices. The options are as follows:

`0` GetNext
This is the default value.

Using this data retrieval option, the system requests one SNMP variable at a time from the device in series, that is, retrieval of one column in a table, one value at a time for a given device. This approach is slower but puts least pressure on the device. In a discovery with multiple entities the expectation is that overall this approach will not slow down the discovery as the SNMP helper is still busy with other activities. This approach might take a long time for individual large devices. This method works with SNMP version 1.

`1` Asynchronous GetNext
Similar to the GetNext method in that one index is retrieved at a time with the difference that all the columns are retrieved in parallel. This is also supported by SNMP version 1 and is faster but it also puts slightly more load on the device.

2 GetBulk
Requests the entire column or multiple columns and individual Get commands in one go. This method requires SNMP version 2 support. If the device only supports version 1 then the retrieval method is broken down into multiple SNMP Get Next and Get commands. This is the fastest retrieval and it does not put much more load on the device than the Asynchronous GetNext method. This method also involves larger packets on the network.

Note: The `Entity.agnt` file, together with all other agent configuration files, can be found in the `$NCHOME/precision/disco/agents` directory.

Discovery agents for wireless networks

Network Manager provides agents that discover devices on wireless networks.

Table 530. Discovery agents on wireless networks

Agent name	Function
Airspace	The Airspace Perl discovery agent retrieves WLAN information from devices using the Airspace MIBs.

Discovery agents on other protocols

Network Manager provides agents that discover devices that use other protocols than ones previously described.

Note: Before enabling these agents, configure SNMP access and the SNMP Helper.

Table 531. Discovery agents on other protocols

Agent name	Function
AlteonStp	This is a Spanning Tree Protocol discovery agent for Alteon switches that support the dot1dStp section of the BRIDGE-MIB.
BrocadeFDPSnmp	The `BrocadeFDPSnmp` agent provides Layer 2 links using the Foundry Discovery Protocol (FDP). The agent establishes links between Brocade devices. By using the `FOUNDRY-SN-ROOT-MIB` and `IF-MIB MIB` files, the agent can discover the neighboring devices and store minimal information about the local device and its corresponding neighbor. This agent uses the index from the FDP network layer address of the device to find complete information that links to the neighboring devices.
CDP	The CDP agent understands the protocol used among Cisco communication devices. Using CDP, Cisco devices can discover their nearest neighbors and store minimal information about them. This agent begins with the address of a known Cisco device and uses CDP to find more complete information about the locations of other connected or neighboring Cisco devices.

Table 531. Discovery agents on other protocols (continued)

Agent name	Function
DefaultLLDP	The DefaultLLDP agent discovers layer 2 connectivity between devices that support the LLDP MIB and have Link Layer Discovery Protocol (LLDP) enabled. Both the LLDP and DefaultLLDP agents use data from the LLDP MIB that is indexed by lldpRemLocalPortNum. This variable indicates which ifIndex or port a particular LLDP connection exists on. The LLDP agent supports devices where lldpRemLocalPortNum refers to the ifIndex on the device: typically, Cisco devices. The DefaultLLDP agent supports devices where lldpRemLocalPortNum refers to the port or other arbitrarily assigned index: typically, non-Cisco devices such as Juniper or BNT devices. The DefaultLLDP agent checks if the device supports the Extended-LLDP-MIB. If the device does not support the Extended-LLDP-MIB, lldpRemLocalPortNum is assumed to be a switch port. The agent then uses the dot1dBasePortIfIndex variable from the BRIDGE-MIB to determine the ifIndex of this record. Enable both the LLDP and DefaultLLDP agents so that Network Manager is able to find LLDP connectivity on devices that have different implementations of lldpRemLocalPortNum.
FddiDefault	The FddiDefault agent discovers any device that supports the standard FDDI MIB. When an FDDI device is interrogated, information relating to the interfaces of that device and its upstream and downstream neighbours is returned. The FddiLayer stitcher uses this and all other FDDI agents to determine the FDDI ring topology.
FddiCiscoConc	The FddiCiscoConc agent discovers Cisco Concentrator FDDI devices. Cisco concentrators know the full connectivity of every FDDI ring that passes though them, as opposed to just their upstream and downstream neighbours. Hence the FddiLayer stitcher gives the topology information returned by this agent precedence over that found by FddiDefault.
IEEE8023LAG	Fix Pack 6 The IEEE8023LAG agent discovers the LAG (Link Aggregation Group) Link entities and physical ports associated with the LAG between two network devices. The agent discovers information from Cisco Carrier Routing System (CRS) LAG networks.
LLDP	The LLDP agent discovers layer 2 connectivity between devices that support the LLDP MIB and have Link Layer Discovery Protocol (LLDP) enabled. Both the LLDP and DefaultLLDP agents use data from the LLDP MIB that is indexed by lldpRemLocalPortNum. This variable indicates which ifIndex or port a particular LLDP connection exists on. The LLDP agent supports devices where lldpRemLocalPortNum refers to the ifIndex on the device: typically, Cisco devices. The DefaultLLDP agent supports devices where lldpRemLocalPortNum refers to the port or other arbitrarily assigned index: typically, non-Cisco devices such as Juniper or BNT devices. The LLDP agent checks if the device supports the Extended-LLDP-MIB. If it does, the agent retrieves the mapping between lldpRemLocalPortNum and ifIndex. If the device does not support the Extended-LLDP-MIB, lldpRemLocalPortNum is assumed to be the ifIndex. Enable both the LLDP and DefaultLLDP agents so that Network Manager is able to find LLDP connectivity on devices that have different implementations of lldpRemLocalPortNum.
SONMP	The SONMP agent uses the SynOptics Network Management Protocol, the protocol used between Nortel communications devices. The SONMP agent begins with the address of a known Nortel device and uses SONMP to discover location, containment, address, and connection information from connected, or neighbouring, Nortel devices.

Table 531. Discovery agents on other protocols (continued)	
Agent name	**Function**
StandardSTP	The StandardSTP agent discovers STP connectivity data on any STP-enabled switch that supports the dot1dSTP section of the BRIDGE-MIB. You should run this agent in addition to any other necessary switch agents in order to discover STP backup (blocking) connections. The STP switch discovery method has the following advantages over other switch-based discovery methods: • Hidden links: STP backup (blocking) connections are discovered. • Speed: the agent completes in Phase 1; no pinging is required. **Note** : The STP agent only shows connections between STP enabled switches, that is, it ignores connections to nodes, non-switch devices, and non-STP enabled switches. This agent will not discover multiple STP instances, VLANs, or Virtual Routers.

Context-sensitive discovery agents

There are several agents that take part in a context-sensitive discovery.

Attention: Enabling a context-sensitive discovery enables all the Context agents. The discovery process runs the appropriate agents for the devices to be discovered, based on the OID defined in the agent definition files. Disabling a context-sensitive discovery disables all the Context agents and no Context agents will run. You do not need to enable or disable individual Context agents.

Note: These agents require Telnet access and the Telnet Helper.

Table 532. Context-sensitive discovery agents	
Agent Name	**Function**
CheckpointContext	This Perl agent queries CheckPoint VSX firewalls to retrieve context data. Retrieving context data allows other context-sensitive discovery agents to retrieve interface and connectivity data from the appropriate contexts.
CheckpointVSX	The CheckpointVSX agent retrieves details of the virtual firewalls running on a VSX device. For those virtual firewalls it retrieves further information to resolve the dependencies between the physical hardware and the logical interfaces running on top of the hardware. This information is then used to inform the root cause analysis and better model the VSX devices.
CiscoNexusContext	Discovers VRF context-sensitive information from Cisco Nexus family devices. Prerequisite: You must configure an SNMP context for each VRF so that the VRFs can be discovered. The SNMP context is used to discover the IP address and IP routing data from non-default VRFs.
RedbackContext	The RedbackContext agent discovers virtual router context-sensitive information for Redback devices.

Table 532. Context-sensitive discovery agents (continued)	
Agent Name	**Function**
UnisphereERXContext	The UnisphereERXContext agent discovers virtual router and VRF context-sensitive information for Juniper ERX devices. You can restrict the scope of the VRF contexts discovered by configuring the optional DiscoAgentDiscoveryScoping section in the .agnt file. The configurable options are: • IncludeVRF – allows the discovery of the named VRF. • ExcludeVRF – does not discover the specified VRF. VRF names are case-sensitive. The wildcard " * " can be used in place of a VRF name to apply the filter to all VRFs. If no filters are specified, all VRFs will be discovered by default.

Task-specific discovery agents

There is a group of discovery agents that are task-specific.

Table 533. Task-specific discovery agents	
Agent name	**Function**
AlliedTelesynATSwitch	The AlliedTelesynATSwitch agent discovers Ethernet switches made by Allied Telesyn. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.
AlteonSwitch	The AlteonSwitch agent retrieves layer 2 connectivity information from Alteon load balancers and Ethernet switch modules. **Note:** Configure SNMP access and the SNMP Helper before enabling this agent.
ARPCache	The ARPCache agent assists in populating the Helper Server with IP to MAC address mappings in preparation for the Ethernet-based discovery agents. You must run this agent if you are running a layer 2 discovery. This agent is optional if you are running a layer 3 discovery. However, it can be more efficient to use the ARP Cache discovery agent because in most network environments the ARP helper can only run on one subnet at a time. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.

Table 533. Task-specific discovery agents (continued)

Agent name	Function
ASM	Determines whether ASMs for the following commercial server and database products are running on a device: • Oracle • Apache • Microsoft SQL Server • Microsoft Exchange • Microsoft Internet Information Server (IIS) • Microsoft Active Directory • IBM WebSphere • BEA WebLogic • SAP • Sybase ASE • IBM Lotus® Notes/Domino Server The ASM agent determines whether an application is running by querying ASM-specific MIBs for the device. These MIBs are installed by default when you install Network Manager. The ASM agent can only retrieve this information from network devices on which ASM is deployed. Typically, you would deploy a ASM sub-agent on each commercial server and database product which is running on a device and whose performance you wish to monitor.
BGPPeerNextHop Interface	All PE to CE interfaces are added to a members list and an event on any of the interfaces in this members list causes the system to generate a synthetic MPLS VPN SAE. This agent, which is off by default, enables the generation of MPLS VPN service-affected events (SAEs) based on interfaces dependencies deeper in the core network. This agent calls the AddLayer3VPNInterfaceDependency.stch stitcher. This stitcher determines all PE to core provider router (P) interfaces and P to PE interfaces involved in a VPN. These PE -> P and P ->PE interfaces are added to a dependency list. An event on any of the interfaces in this dependency list causes the system to generate a synthetic MPLS VPN SAE. If an MPLS VPN SAE has already been generated based on an event on any of the interfaces in the members list, then any events in interfaces in the dependency list will be added as related events to that already generated MPLS VPN SAE.
CiscoNexusVdc	Discovers Virtual Device Context (VDC) information from Cisco Nexus 7000 and 9000 series devices. Each VDC instance is hosted under a hypervisor entity contained by the physical device.

Table 533. Task-specific discovery agents (continued)	
Agent name	**Function**
CiscoVoIPHttp	The CiscoVoIPHttp agent is a Perl-based agent that extracts device data and network information about the Cisco VOIP device. The agent uses a Network Manager-specific Perl module to retrieve the information from the following pages of the device: • `http://`*`IP ADDRESS`*`/CGI/Java/Serviceability?adapter=device.statistics.device` • `http://`*`IP ADDRESS`*`/CGI/Java/Serviceability?adapter=device.statistics.configuration`
CM	Retrieves data from cable modems that are connected to a cable modem termination system device. **Note**: If activated, this agent retrieves a large amount of information. Activating this agent may therefore place a heavy load on memory. You should only activate this agent if specific cable modem information is required beyond that provided by other agents.
CMTS	Discovers cable modem termination system devices. This agent also discovers cable modem connectivity. **Note**: If activated, this agent retrieves a large amount of information. Activating this agent may therefore place a heavy load on memory. You should only activate this agent if specific cable modem information is required beyond that provided by other agents.
ExtraDetails	The ExtraDetails agent is a text-based agent that builds on the basic SNMP information already retrieved by the Details agent.This agent retrieves the following information: • sysDescr • sysLocation • sysUpTime • sysServices • ifNumber **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.
HPNetworkTeaming	The HPNetworkTeaming agent discovers secondary NICs on HP Proliant Teamed network cards.If this agent is not enabled, only the primary NIC on an HP Proliant device will be discovered (as a local neighbour to the server) because only this NIC resides in the `ifTable`. This agent will create all NICs as local neighbours to the server. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.

Table 533. Task-specific discovery agents (continued)

Agent name	Function
LoopbackDetails	The LoopbackDetails agent is used to ensure that the management interface of a device is used in the topology and in subsequent monitoring as the main IP/name combination. The agent retrieves information needed to identify the management interfaces. This data is then used in the NamingFromLoopbackDetails stitcher. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.
MACFromArpCache	The ArpCache agent must be enabled for this agent to run. The MACFromArpCache agent is optionally activated in phase 3 of Discovery. It uses the ArpCache information retrieved by the ArpCache agent to retrieve the MAC address of the device. The agent is useful as it does not require SNMP access to the device to obtain the MAC address.
MACFromTDWDatabase	This agent queries the Tivoli Data Warehouse. The agent retrieves the mapping of MAC address to IP address for each server. Run this agent if the connectivity of servers is not represented properly in the network topology due to incomplete mapping of MAC address to IP address of the servers. The IBM Tivoli Monitoring (ITM) Operating System (OS) Agent must be running on the servers in the network for which you want to retrieve information. You must also have access to the Tivoli Data Warehouse database in order to access the information retrieved by the ITM OS agents. Before running this agent, add the access details for the Tivoli Data Warehouse database to the DbLogins.cfg file. Edit the DbLogins.cfg file and add an insert similar to the following example: `insert into config.dbserver ( m_DbId, m_Server, m_DbName, m_Schema, m_Hostname, m_Username, m_Password, m_PortNum, m_EncryptedPwd, m_OracleService ) values ( "TDW", "oracle", "someDbName", "someSchema", "serverHostName", "username", "password", port, [0|1], 1 );`

Table 533. Task-specific discovery agents (continued)	
Agent name	**Function**
NetScreenArpCache	The NetScreenArpCache agent retrieves information from ARP tables configured in Netscreen devices and processes the tables to obtain the IP to MAC translation. The agent then sends the ARP information to the ARP Helper. After further processing, the ARP Helper sends the IP and MAC address mapping to the ARPHelperTable. The NetScreenArpCache agent uses the SNMPv2-SMI Standard MIB. **Note:** The ArpCache agent does not process the Netscreen devices processed by the NetScreenArpCache agent. This is to avoid conflict in the ipForwarding value as Netscreen is recognized as a non-routing device by the ArpCache agent.
NMAPScan	The NMAPScan agent is a Perl agent that runs the NMAP scanner against devices discovered by Network Manager. By default, the agent runs against devices that do not have SNMP access, or devices that have SNMP access but return sysObjectIds of devices from Apple, Compaq, IBM, Microsoft, Sun, Network Harmoni, UC David, Net-SNMP, and HP. The agent retrieves the following data: • Operating System Fingerprint details • TCP/UDP port and application information including port number, name, state, type, and service You must install NMAP version 4.85 or later on the same server where the Network Manager core components are installed. You must then edit the `NMAPScan.pl` file and specify the path to the NMAP binary in the **`my $nmapBinary`** line, and remove the comment from the beginning of the line. NMAP is available at http://nmap.org. **Attention:** Enabling the NMAPScan agent can extend the duration of the discovery. NMAP has a large number of scan options, refer to the NMAP documentation for more information. The following options are set by default for NMAP: • -sS: Perform a TCP SYN scan • -sV: Enable service version identification • -PN: Do not ping each target (Network Manager already uses the ping or file finder, or both) • -O: Enable Operating System fingerprinting • -oX: Enable XML output **Important:** Do not change this value.
OSInfo	Retrieves information about the operating system running on discovered devices. This agent only runs against Cisco and Juniper devices. The agent retrieves the following information: • OSType • OSVersion • OSImage

Table 533. Task-specific discovery agents (continued)	
Agent name	**Function**
SSM	The SSM agent retrieves MIB information by SNMP from devices running SSM agents. This agent retrieves information such as the software installed on the device, running processes, CPU utilization, storage devices on this entity, free disk space, and so on. The SSM agent can only retrieve this information from network devices on which the SSM Agent is deployed. Typically, you would deploy a SSM Agent on devices whose performance you wish to monitor. For more information on the SSM Agent, see the *SSM Application Guide*. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.
SSMOracle	The SSM application and the Oracle monitoring package must also be running. The SSMOracle agent retrieves MIB information by SNMP from devices running SSM agents. This agent retrieves information such as the Oracle database names, fields, and database sizes. The SSMOracle agent can only retrieve this information from network devices on which the SSM Agent is deployed. Typically, you would deploy a SSM Agent on devices whose performance you wish to monitor. For more information on the SSM Agent, see the *SSM Application Guide*. **Note:** Before enabling this agent, it is necessary to configure SNMP access and the SNMP Helper.
TunnelAgent	Template for a Perl agent to retrieve information about all tunnels, including IPv6 over IPv4 tunnels, present in the network. This agent works in conjunction with the IPv6Interface agent.

Discovery agents for IPv6

Network Manager provides a Perl agent template that you can use as a base for developing Perl agents to retrieve IPv6 interface data.

Table 534 on page 845 describes the Perl agent templates.

Note: Rather than having agents with specific IPv6 capabilities, most of the discovery agents have IPv6 capabilities; for example, the InetRouting agent supports IPv6 routing entries but it also downloads IPv4 interfaces and route information.

Table 534. IPv6 agent template	
Agent name	**Function**
IPv6Interface	Template for a Perl agent to retrieve interface information from an IPv6 device. This agent is designed to work in an identical way to the Interface agent. This agent template is located in the Perl agents directory at the following location: `$NCHOME/precision/disco/agents/perlAgents`.

Fix Pack 3 Service Level Agreement agents

Service Level Agreement (SLA) agents retrieve data that relates to the network performance monitoring features of a device.

The agents that retrieve SLA data need SNMP and Ping access to retrieve the data. Before you enable the SLA agents, configure SNMP to enable the agents to access devices and to specify threads, timeouts, and number of retries.

The following table describes the SLA agents:

Table 535. SLA agents

Agent name	Function
CiscoIPSLA	Discovers IP SLA-related data from Cisco devices that support the CISCO-RTTMON-MIB and CISCO-RTTMON-IP-EXT-MIB MIBs. The agent retrieves data such as information on the configured probes.
HuaweiNQA	The Huawei Network Quality Analyser (NQA) agent discovers IP SLA-related data from Huawei NQA devices that support the NQA-MIB MIB. The agent retrieves data such as information on the configured probes.
JuniperRPM	The Juniper Realtime Performance Monitoring (RPM) agent discovers IP SLA-related data from Juniper RPM devices that support the DISMAN-PING-MIB and JUNIPER-PING-MIB MIBs. The agent retrieves data such as information on the configured probes.

Guidance for selecting agents

To discover device technologies (that is, those that use protocols other than IP) on your network, you must ensure that the appropriate agents are active.

The following list provides the non-IP device protocols that are supported by Network Manager. You can select the appropriate agents for these protocols.

- Frame Relay
- Private Network-Network Interface (PNNI)
- Cisco Discovery Protocol (CDP)
- Link Layer Discovery Protocol (LLDP)
- Hot Standby Routing Protocol (HSRP)
- Fibre Distributed Data Interface (FDDI)
- Asynchronous Transfer Mode (ATM)
- Integrated Local Management Interface (ILMI)
- Multiprotocol Label Switching (MPLS)

Which IP layer agents to use

The IP layer agents that you need to use depend on the devices on your network:

- If you do not want to have your IP routing tables accessed, you must only use the IpBackupRoutes agent.

  This agent is not used by default as it has the following drawbacks:

  – It retrieves data from a table that is not dynamic. If the router has not been refreshed, then the data retrieved by this agent may be spurious.
  – The table is large and therefore takes a long time to download.

- If there are modern devices on the network, you must use the IpRoutingTable agent and the IpForwardingTable agent.

  These agents provide an accurate picture of IP layer connectivity and are therefore used by default.

Which standard agents to use

The standard agents that you need to use depend on the information you require and the devices on your network.

- The TraceRoute agent can be used if there is a firewall on the network, because SNMP calls cannot always be made through firewalls. If you use the TraceRoute agent, you must specify, as a discovery seed, the subnet node for the subnet on the other side of the firewall.
- The ArpCache agent retrieves the physical address of a device, so is only required (in conjunction with the Switch agents) when performing layer 2 discoveries.
- Frame Relay agents should be run in conjunction with the IP layer agents if you need to add DLCI information to the interfaces of Frame Relay devices.
- Switch agents must be run for a layer 2 discovery.
- The device-specific and protocol-specific agents are only required to discover the devices or protocols to which they relate.

Which specialized agents to run

Several agents need to run only when you need to discover certain device types or network technologies.

The specialized agents that you need to run depend on the devices and protocols in your network:

- The Extreme agent can be used to extract layer 2 connectivity information, EDP neighbors, and VLAN details from Extreme switches.
- The ExtremeESRP agent discovers Extreme Standby Routing Table information from Extreme routing switches.
- The PnniForeSys agent discovers physical ATM connections between devices by using the PNNI (Private Network-to-Network Interface) connectivity information provided by the Marconi ASX series switches.
- The ILMIForeSys agent discovers physical ATM connections between devices by using the ILMI (Interim Local Management Interface) connectivity information provided by the Marconi ASX series switches.
- The CellPath90 agent discovers the ATM connection of a CellPath 90 WAN (Wide Area Network) multiplexer.
- The Marconi3810 agent discovers the Ethernet connectivity of the ES-3810 switches running operating system version 4.x.x.
- The MariposaAtm agent discovers the ATM connectivity of the SE420 and SE440 IADs.

  Note: The Ethernet switching and Frame Relay capabilities of these devices are not currently certified.
- The ILMI agent discovers connectivity between ATM devices running ILMI that support the ATM Forum's ATM MIB. The CiscoPVC agent retrieves PVC data from Cisco devices.
- The AtmForumPnni agent discovers connectivity between devices running ATM Forum PNNI that correctly support the ATM Forum's PNNI MIB.
- For Cisco devices, run the CiscoMPLSSnmp agent if you have the MPLS MIBs enabled on a device, otherwise, use the CiscoMPLSTelnet agent.
- For Juniper devices, run the JuniperMPLSTelnet agent if you wish to discover MPLS paths.
- For Juniper ERX devices (formerly Unisphere), the UnisphereMPLSTelnet agent must be used to discover MPLS paths, as these devices are sufficiently different to the Juniper "M" series routers that a different agent is required.
- The StandardMPLSTE agent discovers MPLS Traffic Engineered (TE) tunnels.
- The StandardIGMP agent discovers networks running the Internet Group Management Protocol (IGMP).

- The StandardIPMRoute agent discovers IP multicasting networks.
- The StandardPIM agent discovers Protocol Independent Multicast (PIM) groups.

Suggested agents for a layer 3 discovery

The recommended agents for a layer 3 discovery depends on your network.

When running a layer 3 discovery, the following agents should be run:

- Details and AssocAddress
- A combination of the following IP layer agents:
  - IpRoutingTable
  - IpBackupRoutes
  - IpRoutingTable and IpForwardingTable
- HSRP
- VRRP
- TraceRoute (if firewalls are present)
- IPv4/6 InetRouting. If you have IPv6 in your network, consider running this agent to discover the connectivity, particularly the IPv6 connectivity.

Tip: Some routers support layer 2 technologies. For example, when an ATM card is located in a router chassis, layer 3 discovery agents, such as the IpRoutingTable agent, only discover interfaces with an IP address. Therefore, to fully discover all the interfaces on routers that support layer 2 technologies, you must run the appropriate agents.

Suggested agents for a layer 2 discovery

The recommended agents for a layer 2 discovery depends on your network.

When running a layer 2 discovery, the following agents must be run:

- Details and AssocAddress
- A combination of the following IP layer agents:
  - IpRoutingTable
  - IpBackupRoutes
  - IpRoutingTable and IpForwardingTable
- Switch
- FrameRelay
- ArpCache
- ATM
- FDDI
- HSRP
- VRRP
- MPLS

Chapter 27. Helper System

The helpers are specialized applications that retrieve information from the network on demand.

Note: If the helpers and the Helper Server are running on a different host to the DISCO process, and these hosts are behind a firewall, then specialized configuration is required to ensure that the Helper System can communicate with DISCO. For more information, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Helpers

Helpers retrieve information from devices and pass it to the Helper Server for retrieval by the agents.

The default helpers are described in the following table.

Table 536. Helpers available with Network Manager

Helper	Executable	Configuration file	Description
ARP	ncp_dh_arp	`NCHOME/etc/precision/DiscoARPHelperSchema.cfg`	Performs IP address to MAC address resolution.
DNS	ncp_dh_dns	`NCHOME/etc/precision/DiscoDNSHelperSchema.cfg`	Performs IP address to device name resolution.
PING	ncp_dh_ping	`NCHOME/etc/precision/DiscoPingHelperSchema.cfg`	Either pings each device in a subnet, an individual IP address or a broadcast or multicast address. The result of the ping could be used to populate the MIB of the device.

SNMP	ncp_dh_snmp	`NCHOME/etc/precision/DiscoSnmpHelperSchema.cfg`
		`NCHOME/etc/precision/SnmpStackSchema.cfg`
		`NCHOME/etc/precision/SnmpStackSecurityInfo.cfg`
			Returns results of an SNMP request such as Get, GetNext and GetBulk.

Table 536. Helpers available with Network Manager (continued)			
Helper	**Executable**	**Configuration file**	**Description**

TELNET	ncp_dh_telnet	`NCHOME/etc/precision/DiscoTelnetHelperSchema.cfg`
		`NCHOME/etc/precision/TelnetStackPasswords.cfg`
		`NCHOME/etc/precision/TelnetStackSchema.cfg`
			Returns the results of an OS command against a specific device using the Telnet or SSH protocol.
			The Telnet Helper supports the following encryption algorithms:
			• 3DES-CBC
			• AES-128-CBC
			• Fix Pack 4 AES-128-CTR
			• Fix Pack 2 AES-192-CTR
			• Fix Pack 4 AES-256-CTR

XMLRPC	ncp_dh_xmlrpc	`NCHOME/etc/precision/DiscoXmlRpcHelperSchema.cfg`	Enables Network Manager to communicate with EMS collectors using the XML-RPC interface.

Helper System operation

At startup, the Helper Server loads up the Helper Server schema from the DiscoHelperServerSchema.cfg configuration file and creates the appropriate helper databases. The Helper Server also creates a Helper Manager for every helper database.

The Helper Manager manages the way in which the helper handles requests from the Helper Server to retrieve network device data. The Helper Manager specifies:

- The request timeout
- The time-to-live for the returned variables
- Whether multiple requests are to be processed in serial or parallel

When the Helper Manager detects a request for network data from the Helper Server, it instructs the associated helper to retrieve the data from the network.

Dynamic timeouts

The Helper System uses dynamic timeouts to handle network requests.

As an example of the benefit of dynamic timeouts, if the SNMP helper is asked to perform numerous SNMP Get requests, the helper might begin to slow down and therefore exceed the timeout. A static timeout would cause the retrieval of data to terminate (with data lost) even though the device is still responding with data.

To prevent this situation, the helpers incorporate a dynamic timeout system in which they note SNMP Get requests and recalculate and update the timeout as the SNMP daemons of the device begin to slow down.

Chapter 28. Discovery stitchers

Stitchers are processes that transfer, manipulate, and distribute data between databases. The discovery stitchers also process the information collected by the agents and using this information to create the network topology.

The discovery stitchers supplied with Network Manager are stored in the following directories and their subdirectories.

- Text-based discovery stitchers (text files with a .stch extension): `$NCHOME/precision/disco/stitchers/`
- Precompiled discovery stitchers : `$NCHOME/precision/platform/`*`platform`*`/lib/`, where *platform* is the operating system on which Network Manager is running.
- dNCIM stitchers: `$NCHOME/precision/disco/stitchers/DNCIM`

For information on stitcher language, see the *IBM Tivoli Network Manager Reference*.

Related reference

Stitchers and stitcher language
Stitchers are pieces of code that are used by different Network Manager processes. They take information from one database, process it, and place the information in its new form in another database, or send the information to another process.

Main discovery stitchers

This topic lists the main discovery stitchers.

The following table describes the main discovery stitchers currently included with Network Manager.

Note: This list is subject to change.

Table 537. List of main discovery stitchers

Stitcher	Function
AddAEPhysicalIFContainment	Adds physical interfaces to the chassis in the Link Aggregation Group (LAG) containment structure of Juniper devices. This stitcher is called by BuildContainment.stch.
AddBaseNATTags	Updates all the private NAT addresses that have a private address with their public address and adds a tag denoting the private address.
AddBasicContainment	Part of the mechanism for containment stitching. This stitcher inserts containment information into the simple chassis.
AddCardContainment	Adds card objects to the `workingEntities.containment` table.
AddChassisIPTag	Preprocessing stitcher that fixes the situation where a device on the network has IP access but the management IP address of that device is not in the IP table of the device. Consequently the device would end up in the NCIM physicalChassis table but with no corresponding entry in the NCIM ipEndPoint table. This stitcher fixes this issue by adding a tag, m_ChassisIPNotInIPTable, which is processed by the ModelNcimDb.cfg file to ensure that the ipEndPoint table is populated.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
AddClass	For each discovered entity, retrieves class definition data from the Active Object Class manager, ncp_class, and adds this data to the workingEntities database. This stitcher is called by the PostLayerProcessing.stch stitcher.
AddEntityContainment	Adds general entity information to the `workingEntities.containment` table.
AddGenericEntityContainment	Processes non-Entity MIB data from the CollectorInventory agent and adds processed data to the workingEntities.finalEntity table. This stitcher is called by the BuildContainment stitcher.
Fix Pack 2 AddGeoLocationData	Adds geographic data to the `m_ExtraInfo` field of the topology record of a device from the `SysLocation` field.
AddGlobalVlans	Builds global Virtual Local Area Network (VLAN) objects using the `translations.vlans` table.
AddIfStackContainment	Adds interface stack objects to the `workingEntities.containment` table.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function

AddLayer3VPNInterfaceDependency	This stitcher determines all PE to core provider router (P) interfaces and P to PE interfaces involved in a VPN. These PE -> P and P ->PE interfaces are added to a dependency list. An event on any of the interfaces in this dependency list causes the system to generate a synthetic MPLS VPN SAE. If an MPLS VPN SAE has already been generated based on an event on any of the interfaces in the members list, then any events in interfaces in the dependency list will be added as related events to that already generated MPLS VPN SAE.
	
	The BGP sessions set up between the PE speakers, and consequently, the VPNs, depend on the PE -> P and P -> PE interfaces for a given VPN and PE pair. The value of adding these interfaces to the VPN dependency list is that it allows the P->PE and PE->P links to be considered in Service Affected Event (SAE) calculations and thus provide a notification that some set of VPNs on a PE are affected by a link problem between PE and P routers.
	
	The diagram below marks with an asterisk the interfaces that the AddLayer3VPNInterfaceDependency stitcher adds as an MPLS VPN SAE dependency. In this diagram, the following conventions are used:
	
	• [ce] is a customer-edge router
	• [PE is a provider-edge router
	• [P] is a provide core router
	
	`[ce]---[PE]*---*[P]---[P]---[P]*---*[PE]---[ce]`
	`|*`
	`|`
	`|*`
	`[PE]---[ce]`
	
	The results of the stitcher manifest themselves as the m_DependsOn list in the following sample record which shows that an example VPN, VPN_CONTAINER_ACME consists of a number of interfaces in the VPN (m_Members list contains the PE->CE facing interfaces) and subsequently depends on the PE->P/P->PE facing interfaces in the m_DependsOn list.
	
	`{`
	`m_Name='VPN_CONTAINER_ACME';`
	`m_Creator='STITCHER CREATED';`
	`m_Description='Logical object for VPN ACME';`
	`m_EntityType=7;`
	`m_ObjectId='VIRTUAL_PRIVATE_NETWORK';`
	`m_HaveAccess=0;`
	`m_IsActive=0;`
	`m_ExtraInfo={`
	`m_VPNName='ACME';`
	`m_MPLSVPNType='MPLS IP VPN MESH';`
	`m_Members=['pe7-cr38.core.eu.test.lab[Vl2]',`
	`'pe7-cr38.core.eu.test.lab[Fa0/3/1]',`
	`'pe8-cr72.core.eu.test.lab[Fa5/0]'];`
	`m_DependsOn=['pe7-`
	`cr38.core.eu.test.lab[Se0/0/0:0.202]',`
	`'pe8-cr72.core.eu.test.lab[Fa0/0]',`
	`'p4-cr28.core.eu.test.lab[Se0/0/1:0.202]',`
	`'p4-cr28.core.eu.test.lab[Gi0/0]'];`
	`};`
	`}`

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
AddLogicalToIpToBaseName	Adds logical information to the `translations.ipToBaseName` table.
AddLoopbackTag	Adds a tag to the ExtraInfo column of the topology database indicating that an interface is a globally addressable loopback interface.
AddLteEntityContainment	Called within the BuildContainment.stch stitcher. This stitcher processes the return records from the CollectorLTE agent and, using this data it builds LTE-related entities within the workingEntities.finalEntity table, as follows: • All LTE-related entities, such as enbFunction, mmeFunction, sgwFunction, and so on. • All lteIinterface entities of various types, such as S1-U , S1-MME , X2 , S5/S8 ,S11. The stitcher also builds the appropriate relationship records among these LTE entities and their respective chassis entities. • Containment relationship records are built in the workingEntities.containment table. • Relationship records of connection and dependency are built in the m_ExtraInfo->m_Members and m_ExtraInfo-> m_Depends member attributes of the appropriate workingEntity record.
AddNoConnectionsToLayer	The final topology layer is constructed by merging the topology information from the various layers. If there is a mismatch in connectivity information provided by the different layers, information from the more detailed layer takes precedence. For example, the Network Layer (layer 3) provides information indicating that a router interface is connected to another router interface. However, information from the more detailed Data Link Layer (layer 2) shows that there is actually a switch between the two router interfaces. The AddNoConnectionsToLayer is used in cases where it is necessary to remove a connection at one layer but keep the connection at a different layer.
AddSwitchRoutingLinks	Adds switch routing data (that assists the RCA plug-in when performing Root Cause Analysis) to the topology database.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
AddTechnologyType	Optional stitcher called by the `PostScratchProcessing.stch` stitcher. This stitcher is commented out by default. If enabled, this stitcher creates a technology type variable for each interface object. This variable can then be used to create technology-based Network Views. See the *IBM Tivoli Network Manager User Guide* for more information about network views. The stitcher creates the technology type variable by adding an `m_Technology` field to the `ExtraInfo` field within the `scratchTopology.entityByName` table for each interface object. The `m_Technology` field is a string, such as `Ethernet`, `ATM`. The stitcher contains a large collection of default technology types; more can be added by directly altering the stitcher. The small processing load associated with activating this stitcher might slow down your discovery slightly.
AddUnconnectedContainment	Gives unconnected entities a default containment. Unconnected entities do not have a parent, except for their main node or interface.
AddVlanContainers	Uses information in the `workingEntities.finalEntity` and `translations.vlans` tables to add VLAN objects to the `workingEntities.containment` table.
AddWLANContainment	Populates `workingEntities.containment` with WLAN containment data.
AddZTEEntityContainment	Invokes either the `AddZTEMSeriesEntityContainment.stch` stitcher or the `AddZTETSeriesEntityContainment.stch` stitcher, based on the device series OID.
AddZTEMSeriesEntityContainment	Builds up the containment information for the ZTE M series device.
AddZTETSeriesEntityContainment	Builds up the containment information for the ZTE T series device.
AdjustedIPLayer	Adjusts the IP layer to move the IP layer connectivity on logical interfaces down to the physical interface for some routers.
AdjustVlanInterfaceContainment	Fix Pack 2 A real world example stitcher to show how containment could be adjusted to meet user requirements. This stitcher is not run.
AgentRetProcessing	Processes data from the `returns` table of each table.
AgentRetToInstrumentationCiscoFrameRelay	Populates the `instrumentation.ciscoFrameRelay` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationFddi	Populates the `instrumentation.fddi` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationFrameRelay	Populates the `instrumentation.frameRelay` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationHSRP	Populates the `instrumentation.hsrp` table with information from the `returns` table of the appropriate agent.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
AgentRetToInstrumentationIp	Populates the `instrumentation.ip` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationName	Populates the `instrumentation.name` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationPnniPgi	Populates the `instrumentation.pnniPeerGroup` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationSubnet	Populates the `instrumentation.subNet` table with information from the `returns` table of the appropriate agent.
AgentRetToInstrumentationVlan	Populates the `instrumentation.vlan` table with information from the `returns` table of the appropriate agent.
AgentStatus	This stitcher sends events to the `disco.events` table about the status of the discovery agents. These events indicate changes in the state of the agent; for example, if it has started, has finished, or has crashed. See also, FinderStatus, CreateStchTimeEvent, and DiscoEventProcessing stitchers.
AnalyseTopology	Analyses a connectivity database to find how many connections there are on each interface.
AnalyseTopologySummary	This stitcher uses the analysis summary information produced by the AnalyseTopology stitcher to provide an optional deeper topology analysis. This functionality is kept separate from the basic topology analysis as it might affect performance or create topology issues on some networks.
AnalyseTopology	Analyses a connectivity database to find how many connections there are on each interface.
AnalyseTopologySummary	This stitcher uses the analysis summary information produced by the AnalyseTopology stitcher to provide an optional deeper topology analysis. This functionality is kept separate from the basic topology analysis as it might affect performance or create topology issues on some networks.
ApplyMainDisplayLabel	Sets the display label for devices in the GUI based on the setting of m_DisplayMode in the disco.config configuration file. Modifies the entities in the workingEntities.finalEntity database table. Called by the BuildFinalEntityTable.stch and RebuildFinalEntityTable.stch stitchers.
ASMAgentRetProcessing	Based on MIB variable data retrieved by the ASM stitcher, this stitcher generates a list of ASM sub agents running on a given device. Each ASM subagent running on a device corresponds to a commercial server or database product running on that device. The list of ASMs enables autopartitioning of devices within a network based on the commercial server or database products running on those devices.
ASAMIfStringLayer	Uses the ASAM ifDescr format to deduce connectivity.
ASMProcessing	Updates entities based on the services running on them.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
ASRetProcessing	Used in MPLS discoveries where devices in different customer VPNs have identical IP addresses. This stitcher performs the processing necessary to differentiate between these devices and correctly resolve device connectivity. This stitcher is called by the AsAgent agent and works with the `ASMap.txt` file in `NCHOME/precision/etc`.
AssocAddressRetProcessing	Processes data in the `AssocAddress.returns` table, sending the device details to the appropriate discovery agent if the device has not already been discovered.
BGPLayer	Builds the BGP layer created by BGP agent. In common with other layer stitchers, this stitcher receives input from relevant agents. This input consists of entity records containing local and remote neighbor data fields. The stitcher uses these records to work out the local and remote connections for each entity.
BuildBaseSubnetRegex	Takes a given subnet and mask and produces a regular expression to find IP addresses in that subnet.
BuildContainment	Calls the following stitchers to add different types of objects to the `workingEntities.finalEntity` table: • `AddBasicContainment` stitcher, which adds device containment information. • `AddCardContainment` stitcher, which adds card containment information. • `AddIfStackContainment` stitcher, which adds interface stack containment information. • `AddEntityContainment` stitcher, which adds general containment information. • `NATAddressSpaceContainment` stitcher, which adds containment information associated with NAT address spaces. • `AddVlanContainers` stitcher, which adds VLAN containment information. You can comment out lines in this stitcher as appropriate in order to exclude types of objects that are not needed. **Note:** This stitcher also manages collector-discovered devices by accepting data from the CollectorInventory agent.
BuildFinalEntity	Builds the records for a single chassis. The BuildFinalEntity stitcher merges data from multiple agents to create the complete definition of an entity. This stitcher is called by the `BuildFinalEntityTable` stitcher.
BuildFinalEntityTable	Uses the entries in the `translations.ipToBaseName` table to populate the `workingEntities.finalEntity` table.
BuildHuaweiVSIContainers	Creates Virtual Switch Instance (VSI) and Virtual Forwarding Instance (VFI) entities and associated containment for Network Provider Edge devices of Huawei VPNs.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function

BuildInterfaceName	Used to control the naming of interfaces. By default, this stitcher is called by the `BuildFinalEntity` stitcher.
	The default naming strategy for any device interface is as follows:
	`baseName[<card>[<port>]]`
	Alternatively Network Manager uses the following default naming convention if the card and port are not valid:
	`baseName[0[<ifIndex>]]`
	You can use the `BuildInterfaceName` stitcher to change the naming convention for an interface in one of the following ways:
	• Specify that you want to use `ifName` or `ifDescr` to name the interfaces rather than the `ifIndex`, card or port information. Using this option, interfaces would have names like the following example:
	`baseName[eth0/0]`
	In this example `eth0` is the `ifName` of an interface.
	To change the naming convention in this way, change the value of `m_UseIfName` in the `disco.config` table.
	• Modify the `BuildInterfaceName` stitcher directly to specify any interface naming convention.

BuildLayers	Activated in the final phase to implement the stitchers that build the layer databases.
BuildMPLSContainers	This stitcher calls the BuildVPNContainers and BuildVRAndVRFContainers stitchers. It builds VPN, VR, and VRF containers.
BuildNATTranslation	Builds a global translation table for all NAT devices.
BuildNexusVRFContainers	Builds Virtual Route Forwarding (VRF) containers for Cisco Nexus devices.
BuildVPNContainers	Creates objects to represent the MPLS VPNs within the system.
BuildVRAndVRFContainers	Creates virtual router (VR) and virtual routing and forwarding table (VRF) objects within the system. These objects are useful for displaying MPLS information.
BuildVSIContainers	Creates Virtual Switch Instance (VSI) and Virtual Forwarding Instance (VFI) entities. This stitcher also creates logical containment of devices associated with VSIs, VFIs, and CE-PE links.
CabletronLayer	Determines connectivity information based on Cabletron data returned by the discovery agents.
CDPLayer	Determines connectivity information based on the data returned by the CDP agent.
CheckAndSendNATGatewaysToArpCache	Sends the NAT gateways to the ArpCache agent.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
CheckForMasterLink	Looks for connections lower down the interface stack that take precedence over connections higher up the stack.
CheckIfMgmtAddress	Determines if a given IP address is a defined management address.
CheckIndirectResponse	Handles indirect ICMP responses due to NAT.
CheckInterfaceStatus	Checks the `ifOperStatus` data and updates the interfaces status where the `ifOperStatus` is not 1.
CheckManagedProcesses	Checks if the processes in disco.managedProcesses have been started, and if they have not been started it attempts to start them.
CheckMultipleIPNoAccess	Checks for devices with no access but multiple IP addresses. Creates interface objects for these IP addresses and updates the entity appropriately.
CheckValidVirtual	Determines if the given IP address is a valid virtual IP address.
CiscoSerialInterfaceLayer	Creates a new layer called CiscoSerialInterfaceLayer connecting Cisco switches that are connected by serial interfaces. By default, the stitcher removes any connections in the CiscoSerialInterfaceLayer that are duplicated in the IPLayer database, to prevent wrong connectivity. The function to remove mesh connections can be turned on or off by editing a flag in the stitcher.
CiscoVSSContainment	CiscoVSSContainment adds new containment entities, representing the two physical chassis and their respective interfaces and objects, to the workingEntities.finalEntity table.
CMTSLayer	Uses the data downloaded by the CMTS agent to build the connection information between cable modem termination systems and the attached cable modem devices.
CollectorAddressTranslation	This stitcher processes devices discovered using an EMS collector. This stitcher performs the following activities: • Ensures that any collector-discovered devices are identified as being the same as the equivalent SNMP-discovered devices. • Stores data on the collector associated with each device. • Performs other administration tasks related to a collector discovery.
CollectorDetailsRetProcessing	This stitcher processes devices discovered using an EMS collector. It processes entries in the returns table of the CollectorDetails agent and sends these entries to other collector discovery agents. The collector discovery agents retrieve detailed device data from the EMS collectors.
CollectorIPLayer	This stitcher builds layer 2 connectivity for devices discovered using an EMS collector based on the connection data supplied by the CollectorLayer2 agent.
CollectorLagLayer	Creates EMS-based Layer 2 connectivity from Alcatel Lucent 5620 Collector Link Aggregation (LAG) information.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
CollectorL1Layer	Uses the data downloaded by the CollectorLayer1 agent to build Layer 1 connectivity information between optical devices and neighboring optical devices. As long as layer 1 data is present, this stitcher will be invoked.
CollectorLTELayer	Processes all return records from the CollectorLTE agent where the variable m_RemoteNbr is not null. Based on information in the m_RemoteNbr variable, the stitcher populates the CollectorLTEControlLayer.entityByNeighbor OQL table to store all LTE control plane connectivity records. For example, the CollectorLTEControlLayer.entityByNeighbor table stores eNodeB to eNodeB connectivity over the X2 interface, and eNodeB to MME connectivity over the S1-MME interface. This stitcher also determines the LTE user plane connectivity from information available in the m_RemoteNbr variable and populates the OQL table CollectorLTEUserPlaneLayer.entityByNeighbor to store all LTE user plane connectivity records. For example , the CollectorLTEUserPlaneLayer.entityByNeighbor table stores eNodeB to Serving Gateway connectivity over the S1-U interface, and Serving Gateway to Packet Data Network Gateway connectivity over the S5/58 interface.
CollectorLTEMMEPool	Builds MME pools in the database. It does this by processing LTE-related records in the workingEntities.finalEntity OQL table and identifying unique MME group identifiers. The stitcher then builds an LTE pool object of type MMEPool in the workingEntities.finalEntity table for every unique MME group. For every LTE pool object of type MMEPool, the stitcher builds an m_MMEList collection list attribute. This attribute stores the entity names of all MMEfunction entities in that LTE pool object.
CollectorLTETrackingArea	Builds a Tracking Area entity in the workingEntities.finalEntity table for every unique tracking area code. It does this by processing LTE related records in the workingEntities.finalEntity OQL table and identifying unique tracking area codes. For every tracking area entity, the stitcher builds an m_CellList collection list attribute. This attribute stories the entity names of all eUtranCell entities in that tracking area.
CollectorRANLayer	Populates the CollectorRANLayer.entityByNeighbor table with logical RAN connections.
CollectorSwitchLayer	This stitcher builds layer 3 connectivity for devices discovered using an EMS collector based on the connection data supplied by the CollectorLayer3 agent.
CreateAndSendTopology	Activates the stitchers that create the topology and send the final Scratch Topology to MODEL. **Note:** If you are using the embedded relational database, dNCIM, instead of the scratchTopology database, then the dNCIM stitchers are called from this stitcher.
ContainResolvableGenericPortEntities	Creates containment for non-ENTITY MIB- like port entities that have existing corresponding interface records.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
ContextAgentRetProcessing	This stitcher is used for context-sensitive discovery data flow. It merges the outputs of all the Context agents for each entity. It then inserts the results of this merge into the `AssocAddress.despatch` table, using the DetailsOrContextRetProcToAgent stitcher.
CreateBGPServices	Creates BGP hosted service entities. A hosted service is a service or application running on a specific device. For example, a device might host BGP and OSPF services. Each BGP hosted service entity describes a BGP process on a router. This stitcher is called by the PostScratchProcessing stitcher following the creation of the scratch topology.
CreateBGPTopology	Creates connections between BGP speakers. These connections are presented in the Network Views, and correspond to working BGP connections at the time of the discovery. This stitcher can also infer BGP peer routers that Network Manager cannot access. These inferred routers might correspond to BGP autonomous systems outside of your company. This stitcher is called by the PostScratchProcessing stitcher following the creation of the scratch topology.
CreateEmsEntities	Creates an element management system (EMS) object for each known EMS in the topology. In a situation where a collector is acting as the EMS itself (that is, the collector does not specify an EMS source) an EMS object is still created to represent the collector. The discovery infers the chassis hosting the EMS if the chassis has not been discovered.
CreateImpactTopology	An optional stitcher that can be used to make a copy of the Scratch Topology before it is sent to the Topology Manager, ncp_model.
CreateIPMRouteRoutes	Manages the creation of upstream and downstream route entities for the routes downloaded from multicast routers. It also aids MDT resolution.
CreateMPLSPE	This stitcher uses BGP information on the customer edge (CE) routers to infer the existence of inaccessible provider edge (PE) routers. The stitcher builds a single object to represent the provider network and each interface represents a BGP peer found on the customer edge device. This allows RCA to be evaluated across the MPLS provider core. Note the stitcher assumes that all out-of-scope BGP peers are members of the provider network. If this is not true then you can configure which specific sections of the network can be considered valid PE IPs and which not. The inferred interfaces will then be limited to those that are within the defined MPLS scope (scope.inferMPLSPEs). Whether an IP is within the MPLS PE scope is determined by the IsInMPLSScope.stch stitcher.
CreateOSPFAreas	Creates and names an OSPF area. This stitcher is called by the PostScratchProcessing stitcher following the creation of the scratch topology.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
CreateOSPFNetworkLSAPseudoNodes	Retrieves data associated with OSPF pseudonodes advertised by designated routers and builds these pseudonodes in the topology. This overcomes the problem of full meshing when representing OSPF area in Network Views and enables connections within OSPF areas to be visualized in a clear, uncluttered manner.
CreateStchTimeEvent	This stitcher sends events to the `disco.events` table about progress within the data processing phase. For example, the stitcher generates events to indicate that the discovery process has started building the working entities table, and that the discovery process has started building the containment table. See also, AgentStatus, FinderStatus, and DiscoEventProcessing stitchers.
CreateVRRPCollection	Creates collections based on the Virtual Router Redundancy Protocol (VRRP) virtual router ID and associated IP address. Called by the PostScratchProcessing stitcher.
CreateTrunkConnections	Modifies the containment model to take account of VLAN trunks.
CreateVlanEntity	This stitcher creates a single VLAN entity object by adding VLAN data to the Scratch Topology.
CreateWLANAP	Populates WLAN Access Points from WLAN agent data.
CreateWLANAPInterfaces	Populates WLAN Access Points interfaces from WLAN agent data.
CreateWLANAPIPLayer	Populates the IPLayer tables from WLAN data.
DetailsOrContextRetProcToAgent	This stitcher is used as part of the context-sensitive discovery data flow. It is equivalent to DetailsRetProcessing but handles context-sensitive discovery. It processes entities from the `details.returns` table, and sends the details to the despatch table of the relevant Context agent.
DetailsRetProcessing	Processes entities from the `details.returns` table, and sends the details to the `AssocAddress.despatch` table.
DetectionFilter	Determines whether a given device passes the detection filter and is to be discovered based on the `detectionFilter` defined in the `scope` database. By default, the discovery filters do not filter out the Network Manager server, because this server usually also serves as the polling station for root cause analysis. In order for root cause analysis to work correctly, the polling station, and hence the Network Manager server, must be part of the topology. If you need to filter out the Network Manager server using the `detectionFilter`, modify the DetectionFilter stitcher and remove the sections of code indicated by comments that prevent the Network Manager server from being filtered.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
DetermineProtocol	Called by other stitchers. This stitcher receives an IP address string as input and based on the contents of the string it determines the associated IP protocol; for example, an input string of: • `1.1.1.1` returns a value of 1, corresponding to the IPv4 protocol. • `2003:3542:45AB::34` returns a value of 3, corresponding to the IPv6 protocol. • `any-old-string` returns a value of 0, corresponding to unknown protocol.
DiscoEventProcessing	This stitcher responds to an insert into the `disco.events` table and creates and sends the appropriate discovery event to the probe for Tivoli Netcool/OMNIbus, nco_p_ncpmonitor process, which then forwards the event to the ObjectServer. You can control whether discovery events are generated by changing the value of the `m_CreateStchrEvents` field in the `disco.config` table. See also, AgentStatus, FinderStatus, and CreateStchTimeEvent stitchers.
DiscoShutdown	Activated when DISCO is shut down. Calls the RefreshDiscoveryTables stitcher.
ExampleContainment1	An example stitcher that could be modified to configure the containment model.
ExampleContainment2	An example stitcher that could be modified to configure the containment model.
FddiLayer	Deduces the FDDI layer topology.
FDPLayer	Determines connectivity between Brocade devices, based on data returned by the BrocadeFDPSnmp agent.
Feedback	Sends device details back to the Ping finder to seed the discovery again.
FinalPhase	Activated in the final phase to implement the final stitchers.
FindAddressSpace	Identifies the address space of an IP address.
FinderStatus	This stitcher sends events to the `disco.events` table about the status of the finders. For each finder, the stitcher sends an event to indicate changes in the state of the finder; for example, if the finder has started, has finished, or has failed. See also, AgentStatus, CreateStchTimeEvent, and DiscoEventProcessing stitchers.
FindGatewayInterfaces	Identifies the gateway interface on NAT translation devices.
FindPhysIpForVirtIp	Used in resolution of HSRP issues. Finds the physical IP address corresponding to a virtual HSRP address.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
FnderProcToDetailsDesp	Processes entries in the `finders.processing` table, and sends the details to one of the following agents: • Details agent, if the device was discovered directly in the network. • CollectorDetails agent, if the record is a device discovered using an EMS collector.
FnderRediscoveryProcessing	Processes data inserted into the `finders.rediscovery` table. Commonly it will cause the device or subnet range inserted into the table to be rediscovered, either immediately or indirectly after verification by the ping finder.
FnderRetProcessing	Processes entities in the `finders.returns` table. Checks whether the device is in scope and moves this entry to the `finders.processing` or `finders.pending` table, depending on whether the discovery is in blackout state.
FullDiscovery	Determines whether a full discovery is to be run.
GetEntityNameByBase	For a given base name and interface index (or interface ID), this stitcher resolves the associated entity name.
GetEntityNameByIp	For a given address and optional address space, this stitcher resolves the associated entity name. An optional base name can also be specified to restrict the search.
GetBaseNameByIp	Returns the base name associated with the supplied IP address, or "" if none is found. If there are multiple matches then the first is used.
HandleIPMRouteDownstream	Processes the IPMRoute downstream routing data for the current device. It creates downstream route entities which populate the ipMRouteDownstream NCIM table. It also tracks end points required by the route, which are created later, and which MDT to associate the route with.
HandleIPMRouteUpstream	Processes the IPMRoute upstream routing data for the current device. It creates upstream route entities which populate the ipMRouteUpstream NCIM table. It also tracks end points required by the route, which are created later, and which MDT uses to associate the route with.
HubFdbToConnections	A precompiled stitcher that processes all of the connections for the Ethernet hubs. It also requires the connectivity information from the Ethernet switch discovery.
IlmiLayer	Creates the ILMI (Interim Local Management Interface) topology connections based upon ATM ILMI information.
InitiateNATGatewayDiscovery	Seeds the Ping finder with the NAT gateway addresses.
IPAddressNaming	Causes the system to name devices using the IP address where the data is valid. This stitcher is optional and is off by default.
IPLayer	Creates the IP layer topology connections.
IpToBaseName	Populates the `translations.ipToBaseName` table with information from the AssocAddress agent.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
IsForcedRediscovery	This stitcher is used to determine if a finder insert is part of a forced rediscovery. Forced rediscovery contrasts with reactive rediscovery, the mode that the Discovery Engine, ncp_disco, adopts after completion of a discovery. In this mode a device is typically only rediscovered if it is new or if the finder insert references a trap, thus suggesting that the entity has been modified. Forced rediscoveries are started using the Discovery Configuration GUI.
IsInMPLSScope	Determines if a given IP address is in the scope of devices considered to be valid CE devices connected to an inaccessible third-party MPLS PE device.
IsInScope	Used by other stitchers to check that an entity is within the scope of the discovery (that is, within the scope defined in the `scope.zones` table). Prior to checking whether an entity is in scope, the stitcher first determines whether the entity is IP or non-IP: • If the entity is an IP-based entity, then scoping is required. The stitcher proceeds to determine if the entity is in scope. • If the entity is non-IP, for example a layer 1 optical device, or a radio access network device, then no scoping is performed. The system assumes that the device is in scope.
LLDPLayer	Determines connectivity information of remote neighbors based on data returned by the LLDP agent. **Note:** If connectivity is incorrectly displayed for a devices, then this might mean that the LLDP MIB on the network device is incorrectly populated. In some cases the relevant MIB data is incorrectly populated with device model number instead of a unique identifier. In this case the LLDP stitcher is unable to calculate LLDP connectivity correctly. To verify that this is the problem, for each of the devices that are not connected correctly you must examine the values of the LLDPChassisId field in the LLDP agent's `LLDP.returns` table. If you determine that the LLDPChassisId field values are not unique, then edit the LLDPLayer stitcher and set the processing method to a value of 2, by changing the following line in the stitcher: `int processingMethod = 2;`
MergeLayers	Merges the layer topologies.
ModifyIPContainment	Modifies the containment of IP interfaces on non-IP forwarding devices so that they are not upwardly connected. This modification is required to trace root cause.
MPLSCE	Tries to resolve CE to PE connectivity for VRF interfaces on a PE where the connecting CE has not been identified. It uses layer 3 information to try to find the correct connectivity.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
MPLSProcessing	The route target-based MPLS post-layer processing is performed. The `MPLSProcessing` stitcher calls the `RTBasedVPNDiscovery` stitcher to perform this processing. The MPLS discovery results in the ability to display an edge view. This stitcher also performs the background processing required to generate service-affected events.
MPLStackProcessing	Ensures that any interfaces that are situated below a VPN supporting interface in the interface stack are marked as being part of the VPNs which flow through the higher interfaces.
NameResolution	Finds entities where the name has not been resolved and attempts to resolve the entity name based on the resolved names of the other interfaces of the device.
NamingFromLoopbackDetails	Provided there is a LoopBack agent running, this stitcher updates the names in the `translations.ipToBaseName` table. The management IP address of the device used by the poll policies is set to one of the loopback addresses, if Network Manager has confirmed that it has SNMP access.
NamingViaManagementInterface	Looks for management IP addresses from the translations.ipToBaseName and ensures the base address and name of an entity is that of the management server.
NATAddressSpaceContainers	Optional stitcher that builds NAT container objects holding devices within a particular address space and creates inserts into the `workingEntities.finalEntity` table for these NAT container objects. Also builds relevant entries into the `workingEntities.containment` table.
NATAgentRetProcessing	Processes the output from the NAT gateway agents.
NATFnderRetProcessing	Performs processing of NAT devices.
NATGatewayRetProcessing	Used in discoveries involving NAT gateways where one or more of the management interfaces of the NAT gateway device is in private address space. This stitcher performs the processing necessary to determine whether each management interface is in public or private address space. This stitcher is called by the NATGatewayAgent agent and works with the `NATGateways.txt` file in `NCHOME/precision/etc`.
NATIpCheck	Resolves an issue where a NAT gateway adds all of its translated IP addresses to its own IP table.
NATTimer	Triggers rediscovery of NAT gateways.
NortelPassportLayer	Resolves the NortelPassport connectivity discovered by the NortelPassport agent.
OSPFLayer	Creates a topology of the OSPF routing within the network. This OSPF routing information is used by the DetermineOSPFDomains stitcher in order to tag devices and interfaces with OSPF domain information.
OspfPostLayerProcessing	Performs OSPF processing after the OSPF topology has been calculated, for example, assignment of OSPF domains.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
ParseASAMIfString	Parses the ASAM Interface description data into its component parts. Called from the ASAMIfStringLayer stitcher.
ParseZyxelIfString	Parses the ZYXEL Interface description data into its component parts. Called from the ZyxelIfStringLayer stitcher.
PeerBasedHuaweiVPLSDiscovery	Identifies the membership and pseudowire interfaces of middle Network Provider Edge and User-facing Provider Edge devices for each Virtual Switch Instance.
PeerBasedPWDiscovery	Used in discovery of enhanced Layer 2 VPNs on an MPLS core network. This stitcher identifies MPLS pseudowire connections retrieved by the Cisco MPLS agents and adds information about these connections to the relevant network entities for viewing in Topoviz. The information is stored as a pseudowire VPN and provides information about the two provider edge (PE) router ends of the pseudowire.
PIMLayer	Creates PIM Topology table based on remote neighbor data from PIM supporting agents. The topology data is used to populate the m_PIMAdjacency data, which in turn is used to populate the PIM Topology in NCIM
PingFinderScopeRefresh	Tells the Ping Finder to refresh its scope. This stitcher is activated by the Discovery Configuration GUI when you refresh the scope, ensuring that the Ping finder has an up-to-date scope.
PnniLayer	Creates the PNNI topology connections provided the connections at both ends have been discovered.
PostLayerProcessing	The PostLayerProcessing stitcher runs stitchers that extend the network model. These stitchers use data from the workingEntities.finalEntity and workingEntities.containment database tables and the topology layers that were created recently in the stitching process. They create entities such as MPLS entities, global VLANs, and switch modules. If you create any custom discovery stitchers, you can run them from the PostLayerProcessing stitcher.
PreProcessIGMPEndPointData	Creates and populates a temporary table consisting of end-point information for each IGMP-enabled interface and known groups. It also tracks the Multicast groups for which there is IGMP data. This data is used by other IGMP stitchers to create end point and group entities.
PresetLayer	Can be used to "preset" undiscoverable connections, if required. This stitcher is not used by default. This stitcher contains advanced configuration settings. Any changes must be made by certified personnel only.
ProcessQinQData	Processes QinQ data associated with interfaces and builds appropriate containment.
ProcessSwitchModules	Identifies which switch modules have their own IP addresses.
ProcRemoteConns	Takes a record containing a remote neighbor and processes remote connections if the agent that discovered it supports indirect connections.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
ProfilingEndFinal ProfilingPhase1 ProfilingPhase2 ProfilingPhase3 ProfilingStartFinal	These stitchers populate the `disco.profilingData` table, providing data on discovery duration, memory usage, and a broad overview of the results of the discovery. This information is used in the estimation of scaling, and provides you with an overview of discovery performance.
PruneSwitchConnections	This stitcher can be used as a way of improving switch connectivity in cases where the switches do not provide full connectivity information. This stitcher is not enabled by default, and must be enabled only on advice from IBM Support.
PVCNamePath	Adds the name of a PVC path to the internal `atmPVCs.memberships` database table.
PVCProcessedRecord	Updates the `atmPVCs` database to indicate which record is currently being processed.
PVCProcessingRecord	Updates the `atmPVCs` database to indicate which record is currently being processed.
PVCTraceAway	Performs PVC tracing.
PVCTraceCrossConnected	Performs PVC tracing.
PVCTracePath	Performs PVC tracing for a given interface using the other PVC tracing stitchers to trace all the paths through the entire ATM section of the network.
PVCTraceTowards	Performs PVC tracing.
RebuildFinalEntityTable	This stitcher is very similar to the BuildFinalEntityTable. It also uses the entries in the `translations.ipToBaseName` table to populate the `workingEntities.finalEntity` table. The difference is that this stitcher is used in rediscovery mode rather than full discovery mode.
RecreateAndSendTopology	This stitcher is very similar to the CreateAndSendTopology.stch. It also activates the stitchers that create the topology and sends the final Scratch Topology to MODEL. The difference is that this stitcher is used in rediscovery mode rather than full discovery mode. **Note:** The dNCIM stitchers, the stitchers that interact with the dNCIM database during the discovery process, are called from this stitcher.
ReDoIpToBaseName	Refreshes the `translations.ipToBaseName` table.
RefreshDiscoveryTables	Refreshes the discovery database tables.
RefreshLayerDatabase	Refreshes a given layer topology database.
RefreshMPLSTEScope	Refreshes the scope of the StandardMPLSTE agent.
RefreshMulticastScope	Refreshes the scope of the StandardPIM agent.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
RelateVirtualDevices	Creates virtual device instances based on Virtual Device Contexts (VDCs) data discovered by the CiscoNexusVdc agent. Links the VDC instances to the physical device through the hostedServices relationship and Hypervisor entity.
RemoveDeviceFromTopology	Removes a device from the topology. The first argument of this stitcher must be the base name of the device to be removed.
RemoveInferredCEDuplicates	When the existence of a CE router is inferred, this stitcher removes potential duplicate devices from the topology.
RemoveOutOfBandConnectivity	Removes connectivity for out of band devices from the fullTopology.entityByNeighbor table.
RemoveWrongConnectionsToTA838	Removes wrong connections from Cisco 7609 and Cisco 3400 to Adtran TA838 devices.
ResetNATMainNodes	Resets the IP of devices whose addresses have been translated by NAT from the private IP we use to resolve connectivity back to the public IP for monitoring. This allows the devices to be connected and visualized correctly and also remain accessible for monitoring purposes.
ResolveHSRPIssues	Checks for entities that have been discovered through their virtual Hot Standby Routing Protocol (HSRP) address. In that situation, the stitcher updates the discovery agent `returns` tables and the `translations.ipToBaseName` to show the correct physical interface.
ResolveHuaweiVPLSConnections	Adds Temporary Wire information for middle Network Provider Edge devices to the data that is modeled by the `ResolveVPLSConnections` stitcher. This stitcher is called by the `ResolveVPLSConnections` stitcher, if discovery of Huawei VPNs is enabled.
ResolveVPLSConnections	Identifies the User-facing Provider Edge interfaces and Pseudowire interfaces of middle Network Provider Edge devices, and far end Network Provider Edges for each VSI. If discovery of Huawei VPNs is not configured, only User-facing Provider Edge interface information is modeled.
ResolveVPLSPortVlans	Uses data returned from the `CISCO-IETF-PW-ENET MIB` to associate pseudo wires with the correct logical interface.
ResolveVRRPAssocAddresses	Resolves issues caused by VRRP addresses. In such a situation, the stitcher updates the discovery agent `returns` tables and the `translations.ipToBaseName` to show the correct physical interface.
RestartDiscoProcess	Calls the **`restart_disco_process.pl`** script which stops the currently running discovery process and starts a new instance of it. It takes a single argument and a new full discovery is initiated by the newly started discovery process if the value is set to 1. If set to 0, then a full discovery is not initiated.
Restitcher	Re-stitches the topology together.

Table 537. List of main discovery stitchers (continued)

Stitcher	Function
RTBasedVPNDiscovery	Discovers MPLS VPNs based on route target usage. This results in an edge view only which shows the MPLS core network with provider edge (PE) routers for VPNs and VRFs within the scope of the discovery. This view does not show the provider (P) routers within the MPLS core network and associated LSPs (label switched paths) that link these P routers. For each PE router discovered, Network Manager holds information on the route targets imported into and exported from that PE router. This enables you to identify which VPNs use which PE routers.
RTBasedVPNResolution	Uses the VRF data pre-processed by the RTBasedVPDiscovery stitcher to resolve VPNs based on Route Target import and export.
ScopeRefresh	Informs the finders and agents that require scope information when the scope table has changed.
SendRelationalTopologyToModel	Sends the relational topology model network traffic detailing the latest discovery.
SendToCollectors	Sends the supplied seed to the Collector finder for rediscover.
SendTopologyToModel	Sends the stitched topology to MODEL.
SerialLinkLayer	Determines connections from the data returned by the SerialLink agent.
SetOSPFServiceDesignatedStatus	Specifies whether or not the router running an OSPF service is a designated router or a backup router.
SONMPLayer	Determines connections from the data returned by the SONMP agent.
SubnetConnections	Creates subnet entities and inserts into each of the interfaces belonging to the subnet. At layer 3 level the interfaces within a subnet are all considered to be connected, so any connections not already discovered are added to the IP layer database.
SubnetToIPLayer	Adds default layer-three containment and/or connectivity.
SRPLayer	Builds the SRP layer to hold the containment information discovered by the SRP agent. In common with other layer stitchers, this stitcher receives input from relevant agents. This input consists of entity records containing local and remote neighbor data fields. The stitcher uses these records to work out the local and remote connections for each entity.
SwitchFdbToConnections	Copies entries from the Switch agent `returns` tables to the `connections` table.
SwitchStpMltProcessing	Adds connections for all links in a multi-link trunk to an entityByNeighbor table.

Table 537. List of main discovery stitchers (continued)	
Stitcher	**Function**
SwitchStpToConnections	Builds a new layer based on the SwitchStp connectivity. Processes the data from the STP agent to create correctly named local and remote entity connection records in the stpTopology database. In common with other layer stitchers, this stitcher receives input from relevant agents. This input consists of entity records containing local and remote neighbor data fields. The stitcher uses these records to work out the local and remote connections for each entity.
SysNameNaming	Causes the system to name devices using the SNMP sysName where the data is valid. This is an optional stitcher that is off by default.
TagManagementInterfaces	Tags the interface that has the IP address used as the main access IP address for a given entity. This stitcher is used in root cause analysis.
TraceRouteConnectivity	Updates the `IPLayer.entityByNeighbor` table with connectivity information retrieved from the TraceRoute agent returns data.
VRFBasedVPNResolution	Uses the VRF data pre-processed by the RTBasedVPDiscovery stitcher to resolve VPNs based on VRF names.
ZTEEnumerationLookup	Provides a lookup table of `enum` type for cards, subcards, slots, NPC, systemId, port, PSU, fan, etc. for `AddZTEMSeriesEntityContainment.stch` and `AddZTETSeriesEntityContainment.stch`.
ZyxelIfStringLayer	Uses the ZYXEL ifDescr format to deduce connectivity.

DNCIM stitchers

The following stitchers all interact with the DNCIM database. The Discovery engine, **`ncp_disco`**, uses these stitchers to store the network model data in the DNCIM database .

The DNCIM stitchers are called from one of the following stitchers:

- `CreateAndSendTopology`: if Network Manager is in full discovery mode
- `RecreateAndSendTopology`: if Network Manager is in rediscovery mode

The following table describes the discovery stitchers that interact with the DNCIM database.

Note: This list is subject to change.

Table 538. List of DNCIM discovery stitchers	
Stitcher	**Function**
`CleanDNCIMCollections`	To conform to the standard model, this stitcher fixes `Collection` by setting the `mainNodeEntityId` to NULL if the `mainNodeEntityId` is not NULL.
`CleanDNCIMObjects`	A placeholder stitcher that calls any stitcher that is required to filter or remove data from DNCIM before it sends the data to MODEL (for example, it calls the `CleanDNCIM_OutOfBandRouterLinks` stitcher).

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
CleanDNCIM_OutOfBandRouterLinks	Removes router link connectivity for out of band devices from the connects table.
CreateInferredChassis	Creates an inferred chassis which usually is a BGP peer, an MPLS PE router visible from a CE router, or an MPLS CE visible from a customer PE.
DeleteDNCIMSubnetCollects	Deletes the collects entries from the subnet objects so that the collects relationship is correctly rebuilt after a partial discovery without deleting the subnet object.
DeleteNodeFromDNCIM	This stitcher is used in partial discovery. It moves a given DNCIM entity into a pending delete domain. If a device is not found after the discovery, then the device is removed entirely. Such devices are removed by the DeleteRemovedEntities stitcher.
DeleteRemovedEntities	The DeleteRemovedEntities stitcher deletes entities that have previously been put aside for deletion during a partial discovery.
GetConnectSpeedOfComponent	Calculates the connection speed of a connection component such as a network pipe (containing multiple paths, hops, and so on through the network). This stitcher iterates over the links resolving the overall connection speed of the component. The stitcher is recursive, so if a component contains subcomponents, then the stitcher is called again on the subcomponent.
GetConnectSpeed	Resolves the connection speed of a simple connection by examining the interface speed of the interfaces at each end.
GetEntityId	Gets the entity ID for a given domain and entity name.
GetInterfaceSpeed	Retrieves the interface speed of an interface.
GetPendingDeleteDomainId	Retrieves the ID of the temporary domain used to store entities affected by the partial discovery. The temporary domain might need to be deleted.
InferDNCIMObjects	Infers the existence of objects from current data. This stitcher replaces functionality in the deprecated PostScratchProcessing stitcher.
PopulateConnectSpeeds	Builds an entry to the connectSpeeds table.
PopulateDNCIM	This is the root DNCIM stitcher, which populates the DNCIM database tables with entity and relationship data.
PopulateDNCIM_BGP	A container stitcher which calls all the BGP-related modelling stitchers as appropriate to model BGP within the topology.
PopulateDNCIM_BGPAutonomousSystem	Models a BGP autonomous system record.

Table 538. List of DNCIM discovery stitchers (continued)	
Stitcher	**Function**
PopulateDNCIM_BGPProtoEndPt	Models a BGP protocol end point.
PopulateDNCIM_BGPServices	Models the BGP services within the system.
PopulateDNCIM_BGPTopology	Builds the BGP topology layer.
PopulateDNCIM_Collect	Builds an entry to the collects table within DNCIM.
PopulateDNCIMCollection	Populates the DNCIM collection table with data.
PopulateDNCIM_Connection	Puts a single connection for supplied arguments into DNCIM connection table.
PopulateDNCIMConnects	Populates the DNCIM connection table with data from layer tables.
PopulateDNCIM_ConnectSpeeds	Populates the connection speed table for a simple connection (no pipes).
PopulateDNCIM_ConnectSpeedsPipe	Populates the connectSpeed entry for a network pipe.
PopulateDNCIM_Containment	Populates the DNCIM containment table with data about the containment relationship, modelling a single individual containment relationship from the arguments passed to the stitcher.
PopulateDNCIMContainment	Transposes the workingEntities.containment table to the DNCIM contains table.
PopulateDNCIM_CreateProbeCollection	Fix Pack 3 Assists with the creation of probe collections.
PopulateDNCIM_CustomGeography	Fix Pack 2 Adds devices to the correct geographical collections based on their location data.
PopulateDNCIMDependencies	Takes the contents of the workingEntities.dependencies table, and adds the relevant relationships to DNCIM.
PopulateDNCIM_Dependency	Populates the dependency table for a single dependency relationship, modelling a single dependency relationship from the arguments passed to the stitcher.
PopulateDNCIMDiscoverySource	Populates the discovery source objects.
PopulateDNCIM_DomainMembers	Populates the domainMembers table within DNCIM.
PopulateDNCIM_Entity	Inserts a record with the given arguments into the DNCIM entityData and domain members tables. This results in the entity showing up in the entity view as this view s based on a JOIN operation of those two tables.
PopulateDNCIMFromEbnbr	Takes the contents of a single discovery entityByNeighbor table, for example, CollectorLayer1, and transfers these contents to the DNCIM connects table.

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
PopulateDNCIM_GeographicLocation	Processes geographical location and region information for devices. Calls the PopulateDNCIM_Entity stitcher to stores the processed location and region information in the DNCIM entityData table. As long as geographical data is supported by the respective vendor EMSs that are interrogated by the LTE collectors, then geographical location data will be determined by the collectors, processed by this stitcher, and the corresponding NCIM topology database tables will be populated with this geographical data. When you create dynamic collection network views of type Geographical Region and Geographical Location, then this geographical data in NCIM is used to create the hierarchical geographical views of the network.
PopulateDNCIM_GlobalVlanUnresolved	Copies a global VLAN entity from workingEntities.finalEntity into the DNCIM globalVlan table.
PopulateDNCIMHostedServices	Populates the hostedService table for the network.
PopulateDNCIM_HostedService	Populates the hostedService table for a single relationship.
PopulateDNCIM_IGMP	Manages the population of DNCIM with IGMP data.
PopulateDNCIM_IGMPService	Creates Multicast IGMP Service entities and populates igmpService.
PopulateDNCIM_IGMPEndPt	Resolves the required IGMP end points and populates igmpEndPoint.
PopulateDNCIM_IGMPGroups	Creates the required group entities and populates igmpGroup. It also makes the Group entities collect any relevant End Points.
PopulateDNCIM_IPMRoute	Models the IPMRoute objects.
PopulateDNCIM_IPMRouteService	Creates Multicast Routing Service entities. The service entities are linked to the appropriate chassis entity as a Hosted Service.
PopulateDNCIM_IPMRouteEndPt	Populates DNCIM with IPMRoute Protocol Endpoint data.
PopulateDNCIM_IPMRouteGroups	Creates the MDT, Group, and Source entities and populates the ipMRouteMDT, ipMRouteGroup, and ipMRouteSource DNCIM tables.
PopulateDNCIM_IPMRouteTopology	Models the topology of the IPM routes

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
PopulateDNCIM_LAG	Manages the population of dNCIM with Link Aggregation Group (LAG) modelling entities, including SAE link aggregation entities (which collect LAG ports), and aggregation group entities (which contains LAG ports).
PopulateDNCIM_LAGEndPt	Creates LAG End Point entities for LAG members.
PopulateDNCIM_LAGLinkCollections	Creates and populates the Link Aggregation Service Affected Event (SAE) entities.
PopulateDNCIM_LTECollection	Populates DNCIM with LTE collection records.
PopulateDNCIM_LTEControlPlane	Populates DNCIM with LTE control plane records.
PopulateDNCIM_LTEDataPlane	Populates DNCIM with LTE data plane records.
PopulateDNCIM_LTEDepends	Populates DNCIM with LTE eUtranSector entities and the corresponding LTE eUtranCell and antennaFunction entities that this eUtranSector entity depends on.

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
PopulateDNCIM_ManagedStatus	Sets the value of the managedStatus field for each of the interfaces of a main node indicating whether that interface should be monitored or not. The tag can take the following values: The managed status of an entity can be one of the following values: **0** Managed state. The entity is managed. A device can be set to managed by using the Topoviz or the Structure Browser GUIs, or by using the ManagedNode.pl or RemoveNode.pl scripts. **1** Unmanaged state. The entity is unmanaged. A device can be set to unmanaged by using the Topoviz or the Structure Browser GUIs, or by using the UnManagedNode.pl or RemoveNode.pl scripts. **2** Unmanaged by ncp_disco. This setting cannot be modified from the GUI. This value is set by the PopulateDNCIM_ManagedStatus.stch stitcher. **3** Unmanaged because the IP address is out of the discovery scope. The device has been discovered through another IP address that is within the discovery scope. A managed status of 3 is usually given to interfaces, rather than chassis. This value is set by the PopulateDNCIM_ManagedStatus.stch stitcher. **Note:** Unmanaged entities do not suppress other events in RCA. The ncp_poller process does not poll unmanaged entities. Events on unmanaged entities have the field NmosManagedStatus set in the alerts.status field in the ObjectServer.
PopulateDNCIM_MPLSTE	Top-level MPLS TE stitcher that calls the other MPLS TE stitchers in order. This stitcher is like the CreateMPLSTE stitcher that does the equivalent work on the scratchTopology database.
PopulateDNCIM_MPLSTEPipeHop	Builds a single MPLS TE pipe segment as part of an MPLS TE tunnel pipe. This stitcher is like the CreateMPLSTEPipeHop that does the equivalent work on the scratchTopology database.
PopulateDNCIM_MPLSTEResources	Builds the MPLS TE Resource objects. This stitcher is like the CreateMPLSTEResources stitcher that does the equivalent work on the scratchTopology database.

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
PopulateDNCIM_MPLSTEServices	Builds the MPLS TE Service objects. This stitcher is like the CreateMPLSTEServices stitcher that does the equivalent work on the scratchTopology database.
PopulateDNCIM_MPLSTETunnelPipe	Builds a network pipe object for an MPLS TE tunnel. This stitcher is like the CreateMPLSTETunnelPipe stitcher that does the equivalent work on the scratchTopology database.
PopulateDNCIM_MPLSTETunnels	Builds the MPLS TE tunnels and LSP objects. This stitcher is like the CreateMPLSTETunnels stitcher that does the equivalent work on the scratchTopology database.
PopulateDNCIM_MXGroupCollection	Populates the Juniper MX group collection object.
PopulateDNCIM_NseEndPt	Populates an NSE end point entity within the topology model to aid in RAN network RCA management.
PopulateDNCIM_OSPFArea	Infers the existence and creates the ospfArea and ospfRoutingDomain objects. This stitcher replaces functionality in the deprecated CreateOSPFAreas and CreateOSPFRoutingDomains stitchers.
PopulateDNCIM_OSPFEndPt	Infers the existence and creates the ospfEndPoint objects. This stitcher replaces functionality in the deprecated CreateOSPFProtocolEndPoints stitchers.
PopulateDNCIM_OSPFLinks	Populates the connects table with OSPF links. This stitcher replaces functionality in the deprecated CreateOSPFPointToPointAdjacencies stitchers.
PopulateDNCIM_OSPFLSANodes	Builds the OSPF LSA nodes within the network as well as their connections.
PopulateDNCIM_OSPFService	Infers the existence and creates the ospfService and ospfNetworkLSA objects. This stitcher replaces functionality in the deprecated CreateOSPFServices and CreateOSPFNetworkLSAPseudoNodes stitchers.
PopulateDNCIM_OSPF	Calls the appropriate stitchers to populate the OSPF objects.
PopulateDNCIM_PIM	Manages the population of DNCIM with PIM data.
PopulateDNCIM_PIMService	Creates Multicast PIM Service entities and populates the pimService table.
PopulateDNCIM_PIMEndPt	This stitcher creates PIM End Point entities and populates the pimEndPoint table.
PopulateDNCIM_PIMTopology	Creates the PIM Topology links.
PopulateDNCIM_PIMNetworkCollection	Creates the PIM Network entity, which collects all PIM routers. It also updates the PIM service display label to indicate the role of the service.
PopulateDNCIM_PipeComposition	Builds and populates the pipeComposition entry for a given relationship.

Table 538. List of DNCIM discovery stitchers (continued)	
Stitcher	**Function**
PopulateDNCIM_ProbeCollection	Fix Pack 3 Creates hierarchical probe collections for use in network views.
PopulateDNCIM_ProbeEndPt	Fix Pack 3 Creates probe end points representing the source or target. Defines the probe service to depend on them, and the probe entity collect them.
PopulateDNCIM_Probes	Fix Pack 3 Creates probe entities representing a network performance probe instance, and contains them beneath the appropriate probe service.
PopulateDNCIM_ProbeService	Fix Pack 3 Creates probe service entities that represent the network probe technology that is supported on the device, and hosts it on the chassis.
PopulateDNCIM_ProbeTopology	Fix Pack 3 Creates links based on probe source or target data, and assigns them to the probe topology.
PopulateDNCIM_ProtocolEndPt	Populates the DNCIM protocolEndPoint table for a single relationship.
PopulateDNCIMProtocolEndPts	Populates the protocolEndPoint table for the topology.
PopulateDNCIM_RanChassis	Populates the DNCIM tables that extend chassis entities with radio area network (RAN) data. The following items are all modelled as chassis entities: • RAN base station (containing transceivers) • RAN base station controller • RAN packet control unit • RAN radio network controller • RAN Node B (containing transceivers and local cells) • RAN service GPRS support node • RAN gateway GPRS support node • RAN mobile switching center server • RAN mobile switching center • RAN media gateway
PopulateDNCIM_RanCollection	Takes RAN area data, defines logical collections from it, and adds devices and cells to the created collections.

Table 538. List of DNCIM discovery stitchers (continued)	
Stitcher	**Function**
PopulateDNCIM_RanContainedElements	Populates DNCIM tables to handle RAN based on the following considerations: • 2G RAN base stations and 3G Node B chassis can contain transceivers. • The transceivers can, in turn, host RAN sectors. • Node B chassis can additionally contain Node B Local Cells.
PopulateDNCIM_RediscoveredEntities	Populates the rediscovered entities table which is used to keep track of partial discovery.
PopulateDNCIM_SLA	Fix Pack 3 Processes SLA-related data for network performance probes.
PopulateDNCIM_SubIfaceStatus	Sets the sub-interfaces to unmanaged state if the m_UnmanagedSubInts option is enable in disco.config and the parent interfaces were set to unmanaged by PopulateDNCIM_ManagedStatus.stch. The stitcher runs recursively to find all the sub-interfaces.
PopulateDNCIMSubnetAndCollects	Models the subnet objects within DNCIM and creates the collects relationship between the subnet and the interfaces on those subnets.
PopulateDNCIMTopologies	Builds the topology model objects within DNCIM with which the presence of a connection within a specific topology is modelled.
PopulateDNCIM_VlanCollections	Creates VLAN connections and creates VLAN collection entities. Called by the InferDNCIMObjects stitcher.
PopulateDNCIM_VTPEdges	Augments VTP domain entities with edge devices connected to VTP entities. This stitcher replaces functionality in the deprecated AddVTPEdges stitcher.
PopulateDNCIM_VTP	Populates the DNCIM vtpDomain table with inferred VTP entities. This stitcher replaces functionality in the deprecated AddVTPCollections.
PopulateDNCIM_WLAN	This stitcher builds the WLAN objects inferred from the topology.
PopulateDNCIM_WLAN80211SpecCollection	This stitcher creates or updates WLAN spec collections.
PopulateDNCIM_WLANAccessPoint	This stitcher populates WLAN Access Points from the discovery engine. The data in the wlanAccessPoint table supplements the chassis data.
PopulateDNCIM_WLANChannelCollection	This stitcher creates or updates WLAN channel collections.

Table 538. List of DNCIM discovery stitchers (continued)

Stitcher	Function
PopulateDNCIM_WLANCollections	This stitcher identifies the entities of interest to various WLAN collections and invokes the stitchers to create or update them.
PopulateDNCIM_WLANServices	This stitcher populates WLAN services from the discovery engine.
PopulateDNCIM_WLANSSIDCollection	This stitcher creates or updates SSID WLAN collections.
PrepareDNCIMForRediscovery	Sets up a temporary domain if necessary, retrieves the domain's ID, and copies the partially rediscovered entities into the temporary domain to make them ready for processing.
RefreshDNCIM	Clears out the DNCIM tables before repopulating the tables.
RemovePendingDomainRelationships	Removes all relationships involving a rediscovered device. The relationships are then recreated by the processing stitchers where the relationships are still present at the time of the rediscovery.
RetrieveDisplayLabel	For a supplied record, retrieves the display label to use in the entityData table.
RetrieveDNCIMEnumeration	For a given group and key, returns the DNCIM enumeration string.
RetrieveDNCIMMapping	Retrieves the mapping value for gven mappingGroup and mappingKey values.
RetrieveDNCIMVendorFromObjectId	Retrieves the vendor value for a given sysObjectId value.
RetrieveDomainId	For a given domain, retrieves the domain ID.
RetrieveEntityId	For a given name and domain ID, retrieves the entity ID. If the ID does not exist in the DNCIM entityName cache, then the stitcher calls the RetrieveNCIMEntityId stitcher to retrieve the entity ID from NCIM and create the same entity ID in the DNCIM entityName cache. This ensures that NCIM and DNCIM entity IDs are consistent where possible. If the RetrieveNCIMEntityId stitcher retrieves an entity ID from NCIM but that entityId is already allocated within dNCIM, then a new entity ID is created in dNCIM and a discovery event is generated with information on the mismatch between the entity IDs in dCNIM and NCIM. If the RetrieveNCIMEntityId fails to communicate with the NCIM database, then it hands back control to the RetrieveEntityId stitcher. The RetrieveEntityId then generates a new entity ID in the dNCIM entityName cache and generates a discovery event indicating that the NCIM database is down.

Table 538. List of DNCIM discovery stitchers (continued)	
Stitcher	**Function**
`RetrieveNCIMEntityId`	Called by the RetrieveEntityId stitcher to requests an entity ID from the NCIM database if it has not been possible to retrieve the entity ID from the DNCIM entityName cache. If the ID does not exist in NCIM, then the stitcher creates the entity ID in the NCIM database and also creates the same entity ID in the DNCIM entityName cache. This ensures that NCIM and DNCIM entity IDs are consistent where possible. If the RetrieveNCIMEntityId stitcher retrieves an entity ID from NCIM but that entityId is already allocated within dNCIM, then a new entity ID is created in dNCIM and a discovery event is generated with information on the mismatch between the entity IDs in dCNIM and NCIM. If the RetrieveNCIMEntityId fails to communicate with the NCIM database, then it hands back control to the RetrieveEntityId stitcher. The RetrieveEntityId then generates a new entity ID in the dNCIM entityName cache and generates a discovery event indicating that the NCIM database is down.
`SendDNCIMChangesToModel`	Sends topology changes to ncp_model. The stitcher records the time the topology is sent. Called by PopulateDNCIM.stch.
`UnconnectedToSubnet`	Finds main node entities which are not connected to anything and places them within the appropriate subnet collection.

Cross-domain stitchers

Cross-domain stitchers look for links between devices in different domains and create connections between them in the topology.

The following table describes the stitchers currently included with Network Manager that are used for cross-domain discovery.

Note: This list is subject to change.

Table 539. Cross-domain stitchers	
Stitcher	**Function**
AggregationDomainCollectionOfCollections.stch	Creates collections of collection entities in the aggregation domain.
AggregationDomainCollections.stch	Creates collection entities in the aggregation domain.
AggregationDomainCopyEntity.stch	Creates the entity in the aggregation domain based on the entity in the source domain.
AggregationDomainCreate.stch	Creates an aggregation domain.
AggregationDomainFindEntity.stch	Finds entities within the aggregation domain.
AggregationDomainMain.stch	Updates the aggregation domain after a discovery has finished. Calls the other aggregation domain stitchers.

Table 539. Cross-domain stitchers (continued)

Stitcher	Function
AggregationDomain.stch	Checks that the **ncp_disco** process is not in the Processing phase and then calls the AggregationDomainMain.stch stitcher.
AggregationDomainUpdateChangeTime.stch	Updates the timestamp for collection entities.
AggregationDomainUpdateRequired.stch	Check the timestamps for collection entities to determine if an update is required.
LinkDomains.stch	Controls the linking of domains. You can edit this stitcher to configure how domains are linked.
LinkDomainsActOnInstructions.stch	Processes any instructions held in the linkDomains.instruction table and creates connections via the LinkDomainsCreateConnection stitcher.
LinkDomainsAddInstruction.stch	Other stitchers supply instructions to add cross-domain connections to this stitcher. This stitcher adds the connections to the linkDomains.instructions table, after checking that each connection is not already in the table.
LinkDomainsCheckDNCIMForEntityName.stch	Checks that a specified entityName exists in DNCIM.
LinkDomainsCheckDNCIMValidEntityType.stch	Checks that the specified entityType is valid, and, optionally, that the entityType is of the specified metaclass.
LinkDomainsCreateConnection.stch	Adds the connection to DNCIM.
LinkDomainsCreateEntity.stch	Creates the entity in DNCIM based on the entity in NCIM
LinkDomainsDatabaseSetup.stch	Creates the databases used by the domain linking stitchers.
LinkDomainsGetEntityIdFromDNCIMByEntityName AndDomainId.stch	Checks whether the specified entity is in DNCIM by EntityName and domainId.
LinkDomainsGetEntityIdFromNCIMByEntityName AndDomainName.stch	Checks whether the specified entity is in NCIM by EntityName and domainName.
LinkDomainsGetNumConnectsForEntityName.stch	Retrieves the number of related network elements for an entity.
LinkDomainsLoadInterfaceDescriptionMatches.stch	You can configure cross-domain stitching so that devices are linked to each other based on the interface ifAlias field. You can edit this stitcher to define the interface description matches.
LinkDomainsLoadPresetConnections.stch	You can edit this stitcher to define connections between specific devices.
LinkDomainsPopulateDomainAdjacencies.stch	Populates the domainAdjacencies NCIM database table with the information about the domains that are considered to be adjacent. Adjacent domains are expected to have Layer 2 links between them.
LinkDomainsPreProcessInterfaceMatches.stch	Processes devices with matching interface descriptions.

Table 539. Cross-domain stitchers (continued)

Stitcher	Function
LinkDomainsProcessConnectivityMatrix.stch	Processes devices with matching interface descriptions.
LinkDomainsProcessPresetConnections.stch	Processes devices with preset connections.
LinkDomainsResolveInterfaceToLowestPortDNCIM.stch	Finds the lowest port or interface for a given DNCIM interface entityName.
LinkDomainsResolveInterfaceToLowestPortNCIM.stch	Finds the lowest port or interface for a given NCIM interface entityName.
LinkDomainsViaLayer1NameInterface.stch	Creates connections between domains based on layer 1 connectivity.
LinkDomainsViaLLDP	Creates connections between domains based on LLDP data.
LinkDomainsViaPseudoWires.stch	Creates connections between domains based on pseudowire connectivity.
LinkDomainsViaSlash30Subnet.stch	Creates connections between domains based on /30 subnet connectivity.
LinkDomainsViaBGPSessions.stch	Creates connections between domains based on BGP sessions.
LinkDomainsViaCDP.stch	Creates connections between domains based on CDP data.
LinkDomainsViaMPLSTE.stch	Creates connections between domains based on MPLS TE data.
LinkDomainsViaOSPF.stch	Creates connections between domains based on OSPF data.
LinkDomainsViaOSPFAssist.stch	Switched on if LinkViaOSPF is enabled.
LinkDomainsViaPIM.stch	Creates connections between domains based on PIM data.
LinkDomainsViaUnresolvedFDBPorts.stch	Creates connections between domains based on unresolved Forwarding Database (FDB) ports identified by the switch discovery agents.

Part 5. Administrative reference

Chapter 29. Scripts

Use the supplied Perl, shell, or SQL scripts to perform administration, discovery configuration, product upgrade, or troubleshooting tasks.

Related reference

Discovery agents
Use this information to support the selection of discovery agents to run as part of your discovery.

Administration scripts

Use these scripts to administer domains, manage nodes, query processes, and perform actions on the topology.

AddNode.pl

Use the **AddNode.pl** Perl script to add devices to your network topology.

Description

You might want to add a device to your network topology if you know it has been added since the last discovery.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/AddNode.pl -domain
NCOMS -latency 10000 -debug 4 -verbose 192.168.10.8
```

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/AddNode.pl -domain
NCOMS -file mynodes.txt
```

Command-line options

The following table describes the command-line options for the **AddNode.pl** script.

Table 540. AddNode.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain you want to add the node to.
`-latency` *MessageLatency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-debug` *DebugLevel*	Optional; the level of detail the debugging output provides. Values are 1 to 4, where 4 represents the most detailed output.
`-file` *FileName*	Optional; file containing the list of nodes to be added to the network topology. Add one IP address or host name per line in the file. **Note:** You must provide the names of the nodes either in a file or by entering them in the command line, as described in *host* below.

Table 540. AddNode.pl command-line options (continued)	
Command-line option	**Description**
`-verbose`	Optional; provides more information on the screen.
host	Optional; the name of the node to be added. You can add any number of nodes this way, separated by spaces. The information entered for a node can be either the IP address or the host name. If you do not provide a host name, then the `-file` option must be used.

domain_create.pl

Use the **`domain_create.pl`** Perl script to migrate discovery configuration and existing poll policies from an existing domain to a newly created domain.

Description

If your deployment requires additional network domains, you must configure process control for the domains. Once you have done this, you can then use the **`domain_create.pl`** Perl script to migrate configuration files and network polls from an existing domain to the new domain. You must use one instance of ncp_ctrl to run and manage each domain. The script does not migrate the topology from the original domain.

The script also registers the new domain with the NCIM topology database. To create the connection to the NCIM topology database for the new domain, you must specify the connection details in a domain-specific DbLogins.*new_domain*.cfg configuration file or specify the connection details on the command line.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/domain_create.pl -domain NCOMS2 [ -clone NCOMS1 ]
```

Command-line options

The following table describes the command-line options for the script. All options are optional unless noted as mandatory.

Table 541. domain_create.pl command-line options	
Command-line option	**Description**
`-domain` *New domain*	Mandatory; the name of the domain you have created, for example NCOMS2. **Restriction:** Use only alphanumeric characters and underscores (_) for domain names. Any other characters, for example hyphens (-), are not permitted. **Note:** Provides a warning that the name of the domain you want to create is not all uppercase. If not, the script reads a line from standard input, and if the first character is anything other than y or Y, it exits without creating the domain. If the first character is y or Y, then the script creates the domain.
`-clone` *Existing domain*	The name of the domain to copy. If this option is not specified, a new domain is created with default poll policies and configuration files.
`-help`	Provides help on this command.
`-nopolls`	Configures the script to not migrate polls.
`-password`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The NCIM database password.
`-server`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The type of NCIM database server. Use db2 or `oracle`.
`-host`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The hostname of the NCIM database server.
`-port`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The port of the NCIM database server
`-username`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The username for NCIM database access
`-schema`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The NCIM schema name. By default, this is `NCIM`.
`-dbname`	Mandatory if you are not using a domain-specific DbLogins.*new_domain*.cfg configuration file. The Db2 database name or Oracle Service Name.

Table 541. domain_create.pl command-line options (continued)	
Command-line option	**Description**
`-force`	By convention, domain names are all uppercase. If the name of the new domain (the command-line option to -domain) contains any lowercase letters, then by default, the script prompts to ask if you want to create a domain with that name. The *-force* option suppresses this question, and creates the new domain unconditionally.

domain_drop.pl

Use the `domain_drop.pl` Perl script to remove network domains from the NCIM topology database. The entire topology for the domain is removed, together with any poll policies and network views for that domain. The configuration information for the domain and the topology cache is not affected.

Important: Before you run this script, stop the domain that you want to remove. Use the **itnm_stop** command to stop the domain.

Running the script

The following example shows how to run the script:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/domain_drop.pl -domain obsoletedomain -password
password
```

Command-line options

The following table describes the command-line options for the script.

Table 542. `domain_drop.pl` command-line options	
Command-line option	**Description**
`-clearPolldata`	Optional: Removes data from the poll Data table.
`-domain` *obsoletedomain*	Required: The name of the obsolete domain to remove.
`-force`	Optional: As a safety precaution, you are prompted to confirm that you want to delete the domain. Use the `-force` option to execute this script without receiving the confirmation.
`-help`	Provides help on this command
`-pollsOnly`	Optional: Removes all polling policies and associated information for the domain, but does not remove the domain.
`-transactionSize` *transaction_size*	Optional: To allow cascaded removal of entities from the NCIM database, the entities are removed in a series of transactions, rather than as a single operation. By default, the maximum number of entities to be removed in a single transation is 1000. This option should be supplied only if you encounter problems with the default value.

The following table describes the NCIM topology database command-line options for the script. You can specify these options to override the values in the DbLogins.cfg configuration file.

Note: The options described in the table can be optionally supplied with the following qualifiers:

- `ncim_`: Use this value for accessing NCIM only.
- `ncmonitor_`: Use this value for accessing NCMONITOR only.

For example:

```
-ncim_password ncim -ncmonitor_password ncmonitor
```

Table 543. NCIM topology database command-line options for the `domain_drop.pl` script

Command-line option	Description
`-password` *password*	Optional: The password associated with the domain to remove.
`-server` *db2* | *oracle*	Optional: Type of database server.
`-host`	Optional: Host name or IP address of the device running the DB server
`-port`	Optional: Port number on the host. If this value is not supplied and is not read from the DbLogins.cfg configuration file, then the default port number for the server type is used.
`-username`	Optional: Username used for accessing the database.
`-schema`	Deprecated: Name of the schema. Do not use this option.
`-dbname`	Optional: Database name or Oracle Service Name.

inject_fake_events.pl

Use the `inject_fake_events.pl` Perl script to inject fake events onto specified entities and interfaces in the NCIM topology database.

You can use this script to inject fake events onto entities that match a specified string, together with all the interfaces on those entities. Unless specified otherwise, the script will inject events onto entities of the following types:

- 1: Chassis devices
- 2: Interfaces
- 8: Daughter cards

Alternatively you can specify one or more of the three entity types listed above onto which to inject the fake events.

The script injects two types of events:

- PingFail events
  - Events injected onto chassis entities are always PingFail events. These events have NmosEventMap set to 'PrecisionMonitorEvent.300', where 300 is the precedence value.
  - Events on interfaces are PingFail events if the interface has an IP address. In this case these events have NmosEventMap set to PrecisionMonitorEvent.300, where 300 is the precedence value.
- LinkState events: events on interfaces are LinkState events if the interface does not have an IP address. In this case these events have NmosEventMap set to PrecisionMonitorEvent.910, where 910 is the precedence value.

This result of setting the NmosEventMap value is that the Event Gateway uses only the NmosEntityId to locate the exact entity to which the event pertains.

If you want to inject events onto many entities with very different names, then run the `inject_fake_events.pl` Perl script multiple times using different values for -entityNameString parameter in each case. To make this process easier, run this script multiple times with different arguments using a bash shell script.

Running the script

The following examples show how to run the script:

1. Inject an event onto a single chassis entity named "BakerStreetWAN4".

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -entityType 1
```

2. Inject an event onto a single interface named "BakerStreetWAN4[0 [747]]"

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4[ 0 [ 747 ] ]" -entityType 1
```

3. Inject events onto all matching chassis entities and their interfaces for devices with a name like "BakerStreetWAN4"

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4"
```

4. This example is similar to example 3 but with a -interfaceDescriptionString parameter to restrict the search to FastEthernet interfaces only.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -interfaceDescriptionString "FastEthernet" -entityType 2
```

5. This example is similar to example 4, but using an interface description of "Fa2"

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -interfaceDescriptionString "Fa2" -entityType 2
```

6. This example is similar to example 5, but inject events onto the chassis entities too

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -interfaceDescriptionString "Fa2"
```

7. Create resolution events for those events raised by example 6 by simply adding the -resolution command-line option. Tivoli Netcool/OMNIbus will eventually delete the problem events and their matching resolution events.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -interfaceDescriptionString "Fa2" -entityType 2 -resolution
```

8. To see the SQL queries that are executed, use the -debug 1 command-line option.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -debug 1
```

9. To see the SQL queries and the entities found, use the -debug 2 command-line option.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts
```

```
/perl/scripts/inject_fake_events.pl -domain NCOMS -entityNameString
"BakerStreetWAN4" -debug 2
```

Command-line options

The following table describes the command-line options for the script.

Table 544. `inject_fake_events.pl` command-line options

Command-line option	Description
`-domain` *domainName*	Required: The name of the domain that contains the entities onto which to inject the events.
`-entityNameString` *string*	String used to match names of entities onto which events are to be injected. The script uses this argument to produce an SQL WHERE clause to search for an entityName `LIKE "%`*`string`*`%"` in the field entityName in the NCIM topology database entity view.
`-entityType` *entityType*	Optional: Type of entity onto which events are to be injected. Must be 1, 2, or 8. If this parameter is not specified then events are injected onto entities of all three entity types.
`-interfaceDescriptionString` *string*	Optional: String used to match the ifName and ifDescr fields in the NCIM topology database interface view as a means of further filtering the entities onto which events are to be injected. The script uses this argument to produce an SQL WHERE clause to search for a name `LIKE "%string%"` in the fields ifName and ifDescr in the NCIM topology database interface view.
`-resolution`	Optional: By default, only problem events are injected. The argument -resolution injects resolution events instead of problem events.
`-latency` *latency*	Optional: the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-debug` *number*	Optional: Specify one of the following depending on the debugging detail that you require: • Specify `-debug 1` to see the SQL queries that are executed. • Specify `-debug 2` to see the SQL queries that are executed *and* the devices that are found by the queries.
`-help`	Provides help on this command.

itnm_pathTool.pl

Use the itnm_pathTool.pl script to trace a path between a source and destination device. The script determines the interfaces and physical ports used along the path.

Usage

The script displays the path in ASCII format providing the path is not asymmetric or load-balanced. If the path is asymmetric or load-balanced, the path data is displayed in raw format.

Tracing a path

The following example command line traces a path from IPv4 address 172.16.254.1 to IPv4 address 172.16.2.3.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/webtools/bin/
itnm_pathTool.pl -domain NCOMS -from 172.16.254.1 -to 172.16.2.3
```

Command-line options

The following table describes the command-line options for the script.

Table 545. itnm_pathTool.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory. The network domain in which to perform the path trace.
`-from`	Specifies the source IPv4 address, from which to perform the trace.
`-to`	Specifies the target IPv4 address, to which to perform the trace.
`-delete`	Deletes the specified path from the NCIM topology database.
`-outofband`	You can trace a path between an interface inside your domain and one outside, and this option is referred to as an out-of-band trace. This command-line option forces the use of discovered access IP addresses, if available. **Note:** This reduces ingress accuracy.
`-ping`	Pings each next-hop address to verify it is reachable from the management platform.
`-store`	Stores or updates the retrieved path information in the NCIM topology database.
`-return`	Instructs the path trace to additionally retrieve the return path, from the target device back to the source device. Specifying `-from A -to B -return` is the same as specifying `-from A -to A -via B`. Therefore the command-line options `-return` and `-via` cannot be specified together.

Table 545. itnm_pathTool.pl command-line options (continued)	
Command-line option	**Description**
`-timeout`	Override the default 30 second timeout per prerequisite check.
`-via`	Optional IPv4 address to perform the path trace through. This command-line option cannot be used with the option `-return`.
`-debug` *debug*	The level of debugging output (0-4, where 4 represents the most detailed output).
`-help`	Displays the command line options. Does not start the component even if used in conjunction with other arguments.

ITListener.pl

Use the **`ITListener.pl`** script to listen to messages being passed between processes on the message bus.

Usage

Many Network Manager processes communicate through a message bus. For example, **`ncp_model`** broadcasts topology updates on the message bus. Each process broadcasts on a different subject. For example, **`ncp_model`** broadcasts on the subject MODEL. The **`ITListener.pl`** script listens to messages on the message bus and prints them to screen.

Listening for topology change notifications

The following example command line listens for topology change notifications on the NCOMS domain.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ITListener.pl -domain NCOMS -process Model -messageClass NOTIFY
```

Listening for updates from DNCIM to NCIM

The following example listens for updates from the discovery database DNCIM to the NCIM database.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ITListener.pl -domain NCOMS -process DNCIM2NCIM -messageClase NOTIFY
```

Listening for Network Manager status events

The following example listens for events regarding the status of the Network Manager product.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ITListener.pl -domain NCOMS -process ITNMSTATUS -messageClass NOTIFY
```

Command-line options

The following table describes the command-line options for the script.

Table 546. ITListener.pl command-line options	
Command-line option	**Description**
`-debug` *debug*	The level of debugging output (0-4, where 4 represents the most detailed output).
`-domain` *DomainName*	Mandatory. The discovery domain on which to listen.
`-help`	Displays the command line options. Does not start the component even if used in conjunction with other arguments.
`-subject`	The specific subject to listen to. If you specify a subject, the `-domain` argument is ignored, and the script listens on all domains. If you specify a subject you do not need to specify a messageClass and process. All subjects begin `\'ITNM/\'` and have the domain appended. For example, the **`ncp_model`** notify subject for domain *TESTDOMAIN* is `/ITNM/MODEL/NOTIFY/TESTDOMAIN`.
`-process`	The process to listen to. Valid options are: • Model - ncp_model • Class - ncp_class • Config - ncp_config • Ctrl - ncp_ctrl • Disco - ncp_disco • PingFinder - ncp_f_ping • ITNMSTATUS - status events • DNCIM2NCIM - events passed from the DNCIM discovery database to the NCIM database If you specify a messageClass and process you do not need to specify a subject.
`-messageClass`	The class of messages to listen for. Not all processes support all classes. Classes are: • NOTIFY • QUERY • STATUS If you specify a messageClass and process you do not need to specify a subject.

Fix Pack 1 list_applied_updates.pl

Use the `list_applied_updates.pl` script to check which fixpack and interim fix schema updates have been applied to the NCIM topology database.

Description

The `list_applied_updates.pl` script queries the `ncim.schemaAudit` table and lists the fixpack and interim fix schema updates that have been applied to the NCIM topology database recorded there, in the order that they were applied. Note that if a given fixpack or interim fix was applied but contained no

updates to the schema, it is not recorded in the `ncim.schemaAudit` table, and so this script will not list it.

Note: This script works in conjunction with the `update_db_schemas.pl` script. The `update_db_schemas.pl` script applies the schema changes, and this script is used to check that changes for a particular fixpack or interim release were applied.

Running the script

The script has the following syntax.

- Use a domain name to specify login details:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
list_applied_updates.pl -domain DOMAIN_NAME [ -dbname DATABASE_NAME ] [ -debug ]
```

- Specify login details explicitly:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
list_applied_updates.pl -server DATABASE_TYPE [ -dbname DATABASE_NAME ]
-host DATABASE_HOST -username DATABASE_USERNAME -password DATABASE_PASSWORD
[ -port DATABASE_PORT ] [ -debug ]
```

Command-line options

The following table describes the command-line options for the script.

Table 547. `list_applied_updates.pl` command-line options

Command-line option	Description
`-domain` *DOMAIN_NAME*	The name of a Network Manager domain. This is a convenient way for the script to retrieve the database login details from the relevant `DbLogin.cfg` file. In this case, the options -server, -host, -username, -password and -port are not needed. **Note:** If you have more than one domain that shares the same login details, you only need to run the script for one of the domains. Running the script for the other domains that use those same login details will simply produce the same output.
`-server` *DATABASE_TYPE*	Type of database (Oracle or Db2). This option is not needed if you specified the `-domain` option.
`-dbname` *DATABASE_NAME*	Optional: The service name of the database (or Oracle SID) The default value is NCIM.
`-host` *DATABASE_HOST*	Database server host name. This option is not needed if you specified the `-domain` option.
`-username` *DATABASE_USERNAME*	Username for the database. This option is not needed if you specified the `-domain` option.
`-password` *DATABASE_PASSWORD*	Password for the database user. This option is not needed if you specified the `-domain` option.
`-port` *DATABASE_PORT*	Optional: Database port, if not using the default. This option is not needed if you specified the `-domain` option.
`-debug`	Optional: Prints extra debugging information.

Table 547. `list_applied_updates.pl` command-line options (continued)	
Command-line option	**Description**
`-help`	Optional: Provides help on this command.

ManageNode.pl

Use the **`ManageNode.pl`** perl script to set the status of one or more unmanaged devices back to managed state following a period of maintenance.

Description

You can set the status of one or more unmanaged devices back to managed state following a period of maintenance.

This is useful when a device is in unmanaged state and you want to set it to managed state again to receive alerts that are not tagged unmanaged and are used for root cause analysis.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/ManageNode.pl
-domain NCOMS -user root -pwd fruit -verbose -file mynodes.txt
```

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/ManageNode.pl
-domain NCOMS -user root -pwd fruit -verbose neptune.ibm.com 192.168.0.6
```

Command-line options

The following table describes the command-line options for the script.

Table 548. ManageNode.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory; the name of the domain where the unmanaged node resides.
`-user` *username*	Mandatory; the name of the database user.
`-pwd` *password*	Mandatory; the password for the database user.
`-file` *FileName*	Optional; file containing the list of nodes to be set to managed state. Add one IP address or host name per line in the file. **Note:** You must provide the names of the nodes either in a file or by entering them in the command line, as described in *host* below.
`-verbose`	Optional; provides more information on the screen.
host	Optional; the name of the node to be set to managed state. You can specify any number of nodes this way, separated by spaces. The information entered for a node can be either the IP address or the host name. If you do not provide a host name, then the `-file` option must be used.

ncp_password_update.pl

The `ncp_password_update.pl` script is used to update the passwords that are stored in Network Manager configuration files.

Description

These passwords are used by the Network Manager back-end processes to access the NCIM database and the ObjectServer. The script does not change the NCIM or ObjectServer passwords themselves. However, if your database administrator changes the NCIM or ObjectServer password, then in order to enable the Network Manager back-end processes to access the NCIM database (or the ObjectServer) you must run this script to update the passwords configured in Network Manager to match the new NCIM database (or ObjectServer) passwords.

Note that this script does not change the passwords that are used by the Network Manager GUI components to access the NCIM database and the ObjectServer.

As the `ncp_password_update.pl` script runs, this script requires user confirmation. The following configuration files are affected by the script.

```
DbLogins.cfg
MibDbLogin.cfg
NcoLogin.cfg
```

The changes to the configurations are not automatically detected. For the changes to be detected you must restart Network Manager by using `itnm_stop ncp; itnm_start ncp`. You are prompted for the new passwords during the restart. Original versions of the configuration files are backed up in the directory `$NCHOME/etc/precision/backup/`. All passwords are encrypted when written to the configuration files. This topic details 2 use cases for this script.

- Using the script to update the NCIM password.
- Using the script to update the NCIM and ObjectServer passwords in a failover setup.

Note: The **`set_db_details.pl`** script performs similar database configuration tasks. See the **`set_db_details.pl`** documentation topic for more information.

Running the script

To run the script, use a command-line similar to one of the following use cases.

- Use case 1 - Update only the NCIM password.

  Update only the NCIM password, as the NCIM database password was updated. For example, local security protocols might require that you regularly change all user passwords, including the Db2 user. To update only the NCIM password, use this command-line.

  ```
  $NCHOME/precision/bin/ncp_perl
  $NCHOME/precision/scripts/perl/scripts/ncp_password_update.pl -domain NCOMS -ncim
  ```

- Use case 2 - Update both the NCIM and ObjectServer passwords.

  Update both the NCIM and ObjectServer passwords, as the encryption key on the backup Network Manager domain was updated. For example, in a failover setup the backup Network Manager domain is installed on a separate server to the primary Network Manager domain. After the installation, the encryption key `$NCHOME/etc/security/keys/conf.key` must be copied from the primary domain to the backup domain. Any existing encrypted passwords for NCIM access or ObjectServer access are no longer readable. To update both the NCIM and ObjectServer passwords, use this command-line.

  ```
  $NCHOME/precision/bin/ncp_perl
  $NCHOME/precision/scripts/perl/scripts/ncp_password_update.pl -domain NCOMS
  ```

Command-line options

The following table describes the command-line options for the `ncp_password_update.pl` script.

Table 549. ncp_password_update.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory. The passwords for NCIM or ObjectServer, or both, are updated for this domain and other domains that have the same credentials.
`-ncim`	Optional. No value is needed. Updates the NCIM passwords in the `DbLogins` and `MibDbLogin` configuration files.
`-objectServer`	Optional. No value is needed. Updates the ObjectServer password in the `NcoLogin` configuration file.
`-help`	Optional. Displays help information about the Perl script.

ncp_scan_storm_diagnostic_dir.pl

The Perl script `ncp_scan_storm_diagnostic_dir.pl` is used to purge binary data files from the `$NCHOME/precision/PD/core/storm/` directory, and leave the `javacore.txt` file that gives an indication of the type and source of the problem.

Description

The Apache Storm processes aggregate the Network Manager poll data and are highly resilient. If a problem occurs at run time, the processes restart. When there is a major issue, the java run time can create large files in the `$NCHOME/precision/PD/core/storm/` directory, similar to the following files.

```
core.20150826.164534.6612.0001.dmp
heapdump.20150826.164534.6612.0002.phd
javacore.20150826.164534.6612.0003.txt
Snap.20150826.164534.6612.0004.trc
```

The 3 binary files can be large, and repeated restarts can result in the filling up of disk space. This Perl script `ref_ncp_scan_storm_diagnostic_dir.pl` can be used to purge the directory of the binary data files and to leave only the `javacore` text file, which gives an indication of the type and source of the problem.

Running the script

This script automatically runs when you run storm with `itnm_start`. To run the script, use a command-line similar to one of the following use cases.

- Scan once without logging details.

```
$NCHOME/precision/bin/ncp_perl
$NCHOME/precision/scripts/perl/scripts/ncp_scan_storm_diagnostic_dir.pl
```

- Scan once and log the affected files, if there are affected files.

```
$NCHOME/precision/bin/ncp_perl
$NCHOME/precision/bin/ncp_scan_storm_diagnostic_dir.pl -verbose
```

- Scan every 10 minutes without logging details.

```
$NCHOME/precision/bin/ncp_perl
$NCHOME/precision/scripts/perl/scripts/ncp_scan_storm_diagnostic_dir.pl -loop 600
```

- Scan to see what files might be removed, but the files are not removed.

```
$NCHOME/precision/bin/ncp_perl
$NCHOME/precision/scripts/perl/scripts/ncp_scan_storm_diagnostic_dir.pl -test
```

Command line options

The following table describes the optional command-line options for the `ncp_scan_storm_diagnostic_dir.pl` script. These command-line options are not mandatory.

Table 550. `ncp_scan_storm_diagnostic_dir.pl`command-line options

Command-line option	Description
`-test`	Scan the directory to see what files would be affected, but take no action. This option is non-intrusive, and logs the affected files to `stdout`.
`-verbose`	Log details of any affected files to `stdout`.
`-dir`	Specify a non-default directory to scan. The default directory is `$NCHOME/precision/PD/core/storm/`.
`-loop`	This option can be given with an interval that is specified in seconds, at which to scan the directory. This option is ignored if run with **`-test`**. If you do not want to this script to run at intervals or if you want to run this script as a once off, then omit this option.
`-help`	Optional. Displays help information about the Perl script.

read_ncp_cfg.pl

Use the **`read_ncp_cfg.pl`** Perl script to query the Master Domain Controller process, ncp_ctrl, and extract the current service state of the processes that ncp_ctrl has been configured to run.

Description

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/read_ncp_cfg.pl
```

Command-line options

The following table describes the command-line options for the script.

Table 551. read_ncp_cfg.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain where the Master Domain Controller process resides.

RemoveNode.pl

Use the **RemoveNode.pl** perl script to remove specified devices from the network topology.

Description

The **RemoveNode.pl** script removes the specified devices from the network topology by issuing an OQL command to the Model service to delete the entity from the ncimCache.entityData database table.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/RemoveNode.pl
-domain NCOMS -verbose -file mynodes.txt
```

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/RemoveNode.pl
-domain NCOMS -force 192.168.0.6
```

Command-line options

The following table describes the command-line options for the script.

Table 552. Command-line options

Command-line option	Description
`-debug`*DebugLevel*	The level of debugging output (1-4, where 4 represents the most detailed output).
`-domain` *DomainName*	Mandatory; the name of the domain where the device was discovered.
`-file` *FileName*	Optional; file containing the list of nodes to be removed from the network topology. Add one IP address or host name per line in the file. **Note:** You must provide the names of the nodes either in a file or by entering them in the command line, as described in *host* below.
`-force`	Optional; when used, you are not prompted to confirm the removing of a node.
`host`	Optional; the name of the node to be removed. You can specify any number of nodes in this way, separated by spaces. The information entered for a node can be either the IP address or the host name. If you do not provide a host name, then the `-file` option must be used.
`-latency` *MessageLatency*	The maximum time in milliseconds (ms) that the script waits to connect to another process by means of the messaging bus. This option is useful for large and busy networks where the default settings can cause processes to assume that there is a problem when in fact the communication delay is a result of network traffic.

Table 552. Command-line options (continued)	
Command-line option	**Description**
`-messagelevel`*MessageLevel*	The level of messages to be logged (the default is warn): • debug • info • warn • error • fatal
`-verbose`	Optional; provides more information on the screen.

set_db_details.pl

Use the **set_db_details.pl** script to modify parameters for a database. The script modifies the `DbLogins.`*`DOMAIN`*`.cfg` file.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
set_db_details.pl -domain NCOMS -dbId DNCIM -portNum 2316
```

This example updates the domain-specific `DbLogins.NCOMS.cfg` file and sets the port number for the DNCIM database to 2316. All other settings for the DNCIM database remains unchanged.

Note: The **ncp_password_update.pl** script performs similar database configuration tasks. See the **ncp_password_update.pl** documentation topic for more information.

Command-line options

The following table describes the command-line options for the script.

Table 553. set_db_details.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory; the domain for which database details you want to change.
`-dbId` *Database identifier*	Mandatory; the identifier of the database for which you want to modify parameters.
`-server` *ServerName*	Optional; use this option to change the name of the server the database is on.
`-dbName` *dbName*	Optional; use this option to change the name of the database.
`-schema` *schema*	Optional; use this option to specify the schema to be changed.
`-hostname` *hostname*	Optional; use this option to change the host of the database.
`-username` *user name*	Optional; use this option to change the user name used to log into the database.

Table 553. set_db_details.pl command-line options (continued)	
Command-line option	**Description**
`-password` *password*	Optional; use this option to change the password used to log into the database.
`-portNum` *portNum*	Optional; use this option to change the port number used to access the database.
`-help`	Optional; displays the command line options.

UnmanageNode.pl

Use the **`UnmanageNode.pl`** Perl script to set one or more devices to unmanaged state so that engineers can work on these devices without generating network events. Unmanaged devices are not polled by Network Manager. Events for these devices from other sources are tagged in the **Event Viewer** to indicate they are from an unmanaged device.

Description

If you set a device to unmanaged, polling is suspended for the unmanaged node. In the **Event Viewer**, all alerts are tagged to indicate they are from an unmanaged device, and are not used for root cause analysis. You can also unmanage individual devices or groups of devices from the topology map views. There is also an option to set individual components of a device to unmanaged state using the Structure Browser.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/UnmanageNode.pl
-domain NCOMS -user root -pwd fruit -noMainNodeLookup -verbose -file mynodes.txt
```

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/UnmanageNode.pl
-domain NCOMS -user root -pwd fruit -verbose neptune.ibm.com 192.168.0.6
```

Command-line options

The following table describes the command-line options for the script.

Table 554. UnmanageNode.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory; the name of the domain where the node to be unmanaged resides.
`-user` *username*	Mandatory; the name of the database user.
`-pwd` *password*	Mandatory; the password for the database user.
`-noMainNodeLookup`	Optional; switch that enables both interfaces and main nodes to placed into unmanaged status. Use this option together with the `-file` option. **Note:** If you do not specify the `-noMainNodeLookup` option, then the script only places main nodes into unmanaged state. Where an interface is specified in the file, the script will look up the main node and place that main node into unmanaged status.

Table 554. UnmanageNode.pl command-line options (continued)	
Command-line option	**Description**
`-file` *FileName*	Optional; file containing the list of nodes and interfaces to be unmanaged. Add one IP address or host name per line in the file. **Note:** You must provide the names of the nodes and interfaces either in a file or by entering them in the command line, as described in *host* below.
`-verbose`	Optional; provides more information on the screen.
host	Optional; the name of the node to be unmanaged. You can specify any number of nodes this way, separated by spaces. The information entered for a node can be either the IP address or the host name. If you do not provide a host name, then the `-file` option must be used.

Fix Pack 1 update_db_schemas.pl

Use the `update_db_schemas.pl` script to apply all necessary schema updates for one or more fixpacks or interim fixes.

Description

Starting with Network Manager V4.2, when you download a new fixpack or interim fix, the download includes the `update_db_schemas.pl` and associated update files. These update files include all NCIM topology database schema changes for all fixpacks and interim fixes up to the current fix.

You can bring your NCIM topology database up to date with all schema changes for the current fixpack or interim fix, by running the `update_db_schemas.pl` script. The script will also apply schema changes for multiple fixpacks or interim fixes. For example, if for a particular major release you did not install fixpack 1, but are now installing fixpack 2, running the `update_db_schemas.pl` script will bring the NCIM topology database up to date with all schema changes for both fixpack 1 and fixpack 2.

This script works in conjunction with the `list_applied_updates.pl` script. This script applies the schema changes, and the `list_applied_updates.pl` script is used to check that changes for a particular fixpack or interim release were applied.

Note: Do not run the supplied SQL scripts directly. Running these scripts directly can cause schema errors. To update database schemas, always run the `update_db_schemas.pl` script, which makes the appropriate changes for your system.

For more information on installing fixpacks, see the *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*.

Running the script

The script has the following syntax.

- Use a domain name to specify login details:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
update_db_schemas.pl -domain DOMAIN_NAME [ -dbname DATABASE_NAME ]
[-preview [PREVIEW_FILE] ] [ -debug ]
```

- Specify login details explicitly:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
update_db_schemas.pl -server DATABASE_TYPE [ -dbname DATABASE_NAME ]
-host DATABASE_HOST -username DATABASE_USERNAME -password DATABASE_PASSWORD
```

```
[ -port DATABASE_PORT ] [-preview [PREVIEW_FILE] ] [ -debug ]
```

The following are examples of how to run the script:

1. Preview the schema updates that will be applied.

   ```
   $NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
   update_db_schemas.pl -domain DOMAIN_NAME -preview
   ```

2. Apply the schema updates.

   ```
   $NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/sql/
   update_db_schemas.pl -domain DOMAIN_NAME
   ```

   Note: When the schema updater has successfully applied all the changes for a given fixpack, it writes a row to the `ncim.schemaAudit` table, giving the name of the file that contained those changes and the timestamp when they were applied.

Command-line options

The following table describes the command-line options for the script.

Table 555. `update_db_schemas.pl` command-line options

Command-line option	Description
`-debug`	Optional: Prints extra debugging information.
`-dbname` *DATABASE_NAME*	Optional: The service name of the database (or Oracle SID) The default value is NCIM.
`-domain` *DOMAIN_NAME*	The name of a Network Manager domain. This is a convenient way for the script to retrieve the database login details from the relevant `DbLogin.cfg` file. In this case, the options -server, -host, -username, -password and -port are not needed. **Note:** . If you have more than one domain that shares the same login details, you only need to run the script for one of the domains. Running the script for the other domains that use those same login details has no effect.
`-help`	Optional: Provides help on this command.
`-host` *DATABASE_HOST*	Database server host name. This option is not needed if you specified the `-domain` option.
`-password` *DATABASE_PASSWORD*	Password for the database user. This option is not needed if you specified the `-domain` option.
`-port` *DATABASE_PORT*	Optional: Database port, if not using the default. This option is not needed if you specified the `-domain` option.
`-preview` [*PREVIEW_FILE*]	Optional: Prints the schema changes to be made to a file. By default, this file is located in `/tmp/nm-update.sql`. To give the file a different name, specify the name after the `-preview` option. If you do this, then the preview is written to a file with that name in the current directory.

Table 555. update_db_schemas.pl command-line options (continued)	
Command-line option	**Description**
`-server` *DATABASE_TYPE*	Type of database (Oracle or Db2). This option is not needed if you specified the `-domain` option.
`-updatesDir`	Fix Pack 6 Optional. The location of the schema update files. The default is `/updates/db2/` or `/updates/oracle/`, relative to the directory where the script is run. If that directory does not exist, the script looks in `$NCHOME/precision/scripts/sql/updates/SERVER`.
`-username` *DATABASE_USERNAME*	Username for the database. This option is not needed if you specified the `-domain` option.

Database scripts

Use these scripts to query and control the databases.

catalog_db2_database

Use this script to catalog a Db2 database as part of setting up an existing Db2 database for use with Network Manager.

Running the script

The following example shows how to run the script.

```
$NCHOME/precision/scripts/sql/db2/catalog_db2_database.sh database_name host port
```

Command-line options

The following table describes the command-line options for the **catalog_db2_database** script.

Table 556. catalog_db2_database command-line options	
Command-line option	**Description**
database_name	Mandatory; the name of the database to catalog.
host	Mandatory; the hostname of the server where Db2 is installed.
port	Mandatory; the port of the Db2 server.

configTCR

This script is deprecated as of Fix Pack 11. As of 4.2 Fix Pack 11, Tivoli Common Reporting is not supported. You must use Cognos Analytics. Refer to the Cognos Analytics Knowledge Center at https://www.ibm.com/support/knowledgecenter/SSEP7J.

Running the script

Run this script as the user that installed Network Manager.

The following example deploys reports and configures Oracle data sources.

```
configTCR.sh -d ncim -p netcool -h machine123 -n 1521 -z oracle -b NCIM42
-e ncim -k lk -t /opt/IBM/JazzSM -i install
```

The following example deploys reports and configures Db2 data sources.

```
configTCR.sh -d netcool -p netcool -h machine123 -n 50000 -z db2 -b NCIM42
-e db2inst1 -t /opt/IBM/JazzSM -i install
```

Command-line options

The following table describes the command-line options for the **configTCR** script.

Table 557. configTCR command-line options

Command-line option	Description
-b *database_name* | *service_name*	Optional. The NCIM Db2 database name or NCIM Oracle service name.
-d *password*	The password for the NCIM database. This could be on the local machine or on a remote host.
-e *username*	Optional. The username to use to connect to the NCIM database.
-h *hostname*	Optional. The hostname of the server where the NCIM database is installed.
-i *install*	Optional. Specifies that the network management reports are to be installed. You must use the **install** parameter in all cases after option -i. Used by the Network Manager installation, but also used if you need to install a report package provided in a fix pack.
-j *jazz_admin_name*	The administrator username for Jazz for Service Management.
-n *port*	Optional. The NCIM database port.
-p *jazz_admin_password*	The password for the Jazz for Service Management administrator.
-r *path_to_reports_package*	Optional. If your Network Manager installation has the reports package in a non-default location, you can define where the package is using this option.
Oracle | Oracle -s	Optional. Specifies that the Oracle database name is used instead of using the Oracle service name. By default, service name is used.
-t *jazz_location*	The location where Jazz for Service Management is installed.
-z db2 | oracle	The database server type. Can be db2 or oracle.

create_all_schemas

Use the create_all_schemas script to apply the NCIM schemas to an existing topology database. This script is useful, for example, if an NCIM database was not created during the installation of the product, during an upgrade, or to change from one database type to another. Run the script only after you created the topology database. Otherwise the script fails.

The `create_all_schemas.sh` script requires the following information. Specify the information in the order that is given in the table.

Table 558. Information required by the `create_all_schemas.sh` script

Information	Required or optional	More information
Database type	Required	Specify one of the following values, depending on the database type: • DB2 `db2` • Oracle `oracle`
Database name	Required	Oracle Specify the service name.
Host name	Required	The host name can be the name or the IP address.
User name and password	Required	DB2 Use an existing Db2 user. Ensure that the user is not the root user of the host.
Port number	Required	N/A

Examples

The following example creates the NCIM schemas on an Oracle database that has the service name DB_123 on a remote host that is called samplehost, on port 9088. The user/password combination for connecting to the database is ncim/password.

```
$NCHOME/precision/scripts/sql/create_all_schemas.sh oracle DB_123 samplehost ncim
password 9088
```

create_db2_database

Use this script to create the back end NCIM relational database schemas in a Db2 database.

Running the script

Run this script as the Db2 administrative user.

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/db2/create_db2_database.sh database_name user_name
 [ -force ]
```

Command-line options

The following table describes the command-line options for the **`create_db2_database`** script.

Table 559. create_db2_database.sh and create_db2_database.bat command-line options

Command-line option	Description
database_name	Mandatory; the name of the database.

Table 559. create_db2_database.sh and create_db2_database.bat command-line options (continued)	
Command-line option	**Description**
user_name	Mandatory; the name of the database user that will be used to connect to the database. **Important:** This user must not be the administrative user. This user must be an existing operating system and Db2 user.
-force	Optional; instructs the script to force any existing Db2 users off the instance before attempting to drop the database

create_oracle_database

Use this script to create the back end NCIM relational database schemas in an Oracle database. This script must be run as the Oracle system user.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/oracle/create_oracle_database.sh user_name
password [-asm] [-pdb pluggable_database_name]
```

Command line options

The following table describes the command line options for the **create_oracle_database.sh** script.

Table 560. create_oracle_database command line options	
Command-line option	**Description**
user_name	Mandatory; the Oracle user used to create the ncadmin user. This is usually the system user.
password	Mandatory; the password of the system user.
`-asm`	Optional; include this flag if the Oracle database is using Oracle Automatice Storage Manager (ASM).
`-pdb` *pluggable_database_name*	Required only when running the script with Oracle 12c with RAC. Specifies the Oracle 12c pluggable database name.

create_oracle_ncadmin_user

Use the `create_oracle_ncadmin_user.sh` script to create the `ncadmin` user. Run the script as the sys user, the user with database administrator privileges. The `ncadmin` user is needed to provide access to the dbms_lock methods. These locking methods are used in our stored procedures when creating and dropping partitions associated with historical poll data storage.

This script must be run as the sys user because the script needs to run `sqlplus` as `sysdba`. Also, only the database administrator role has permission to grant `execute on dbms_lock`, which is required for the `ncadmin` user.

Note: If you prefer not to run this script as the sys user, then you can use an alternative method to create the ncadmin user that does not require this script and does not involve the sys user. The alternative method is as follows:

1. Run the commands in the `create_oracle_ncadmin_user.sql` file.

2. As a DBA user, grant execute on dbms_lock to the ncadmin user.
3. As the ncadmin user, execute the commands in the `create_oracle_ncadmin_functions.sql` file.

Both the `create_oracle_ncadmin_user.sql` and `create_oracle_ncadmin_functions.sql` SQL files can be found in the same directory as the `create_oracle_ncadmin_user.sh` script.

Running the script

As the `sys` user, run the `create_oracle_ncadmin_user.sh` script on the server where the database is installed. Run the script as in the following example.

`./create_oracle_ncadmin_user.sh` *`user password`* `[-pdb` *`pluggable_database_name`*`]`

Command line options

The following table describes the command line options for the **`create_oracle_ncadmin_user`** script.

Table 561. create_oracle_database command line options

Command-line option	Description
user_name	Mandatory; the Oracle user used to create the ncadmin user. This is usually the sys user.
password	Mandatory; specifies the password of the sys user.
pluggable_database_name	Specifies the Oracle 12c pluggable database name. If you are running Oracle 12c with RAC, you must use a pluggable database.

drop_db2_database

Use this script to remove the back end NCIM relational database implemented using Db2.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/db2/drop_db2_database.sh database_name [ -force ]
```

Command-line options

The following table describes the command-line options for the **`drop_db2_database`** script.

Table 562. drop_db2_database.sh and drop_db2_database.bat command-line options

Command-line option	Description
database_name	Mandatory; the name of the database.
-force	Optional; instructs the script to force any existing Db2 users off the instance before attempting to drop the database

drop_oracle_database

Use this script to remove the back end NCIM relational database implemented using Oracle. This script must be run as the Oracle system user.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/oracle/drop_oracle_database.sh user_name
password [-pdb pluggable_database_name]
```

Command-line options

The following table describes the command-line options for the **drop_oracle_database** script.

Table 563. drop_oracle_database.sh and drop_oracle_database.bat command-line options

Command-line option	Description
user_name	Mandatory; the name of the Oracle database user. This is usually the system user.
password	Mandatory; the password of the database user.
-pdb *pluggable_database_name*	Required only when running the script with Oracle 12c with RAC only. Specifies the Oracle 12c pluggable database name.

populate_db2_database

Use this script to populate the back end NCIM relational database schemas in a Db2 database. You usually run this script after having created the NCIM relational database using the shell or batch script `create_db2_database`.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/db2/populate_db2_database.sh database_name user_name
 password [ -force ]
```

Command-line options

The following table describes the command-line options for the **populate_db2_database** script.

Table 564. populate_db2_database.sh and populate_db2_database.bat command-line options

Command-line option	Description
database_name	Mandatory; the name of the database.
user_name	Mandatory; the name of the database user.
password	Mandatory; the password of the database user.
-force	Optional; instructs the script to force any existing Db2 users off the instance before attempting to drop the database

populate_oracle_database

Use this script to populate the back end NCIM relational database schemas in an Oracle database. You usually run this script after having created the NCIM relational database using the shell or batch script `create_oracle_database`. This script must be run as the ncim user, the user created by the **`create_oracle_database.sh`** script.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/oracle/populate_oracle_database.sh
 user_name password [-pdb pluggable_database_name]
```

Command-line options

The following table describes the command-line options for the **`populate_oracle_database`** script.

Table 565. populate_oracle_database.sh and populate_oracle_database.bat command-line options

Command-line option	Description
user_name	Mandatory; the name of the database user. This is usually ncim.
password	Mandatory; the name of the database.
-pdb *pluggable_database_name*	Required only when running the script with Oracle 12c with RAC only. Specifies the Oracle 12c pluggable database name.

restrict_oracle_privileges.sh

This script applies to NCIM databases created using Oracle as RDBMS. This script is typically run once all Oracle databases and schemas have been created. At that point this script can be used to revoke database creation privileges from the NCIM database user. Only those operations that are required for Network Manager during run time will remain granted.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/oracle/restrict_oracle_privileges.sh user_name
 password [-pdb pluggable_database_name]
```

Command-line options

The following table describes the command-line options for the **`restrict_oracle_privileges.sh`** script.

Table 566. restrict_oracle_privileges.sh command-line options

Command-line option	Description
user_name	Mandatory; the name of the database user. This is usually the system user.
password	Mandatory; password of the database user.
-pdb *pluggable_database_name*	Required only when running the script with Oracle 12c with RAC. Specifies the Oracle 12c pluggable database name.

uncatalog_db2_database

Use this script to uncatalog a Db2 database if you have changed the hostname, port, or database name.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/sql/db2/uncatalog_db2_database.sh database_name
```

Command-line options

The following table describes the command-line options for the **uncatalog_db2_database** script.

Table 567. uncatalog_db2_database command-line options

Command-line option	Description
database_name	Mandatory; the name of the database to uncatalog.

Discovery scripts

Use these scripts to query and control the discovery.

Discovery agent Perl scripts

Some discovery agents are implemented using Perl scripts, and include the following discovery agents. All of these agents are located in `$NCHOME/precision/disco/agents/perlAgents`.

- AlcatelVRRP.pl
- Entity.pl
- NATTextFileAgent.pl
- ASAgent.pl
- iprouting_inperl.pl
- NortelPassport.pl
- CiscoSwitchInPerl.pl
- IPv6Interface.pl
- OSInfo.pl
- NATGatewayAgent.pl

audit.pl

Use the **audit.pl** script to generate a status report for the Discovery engine, ncp_disco. The output is in html format and reports information about discovery, agents and stitchers. You can set the frequency at which the status report is generated. The default is 500 seconds.

Description

This script produces a status report for the Discovery engine, ncp_disco. The file containing this report is output to the following location:

```
current_directory/audit.html
```

The status report includes information on the current state of each of the following:

- Discovery mode
- Discovery phase

- Blackout state
- Cycle count

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/audit.pl
-domain NCOMS [ frequency ]
```

Command-line options

The following table describes the command-line options for the script.

Table 568. audit.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain on which the discovery was run.
frequency	Optional; the frequency, in seconds, at which the report generated by the audit script is run. The default is 500 seconds.
`-debug` *debug_level*	Optional; specifies required debug level.
`-latency` *latency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-messageLevel` *messageLevel*	Optional: The level of messages to be logged (the default is warn): • debug • info • warn • error • fatal

Related reference

disco.status table

Use the disco.status table to monitor the progress of the **ncp_disco** process during the discovery process.

BuildSeedList.pl

Use the **BuildSeedList.pl** Perl script to write a list of seeds to a file.

Description

The **BuildSeedList.pl** Perl script retrieves the list of hostnames and IP addresses discovered during the discovery and writes this list to a file. The script also provides the insert that can be used in the file finder to use this list to seed the discovery. The use of a fully populated seed list for discovery speeds up discovery time.

The file containing the list of hostnames and IP addresses discovered during the discovery is output to the following location:

```
$NCHOME/etc/precision/seedfile.txt
```

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
BuildSeedList.pl
```

Command-line options

The following table describes the command-line options for the **BuildSeedList.pl** script.

Table 569. BuildSeedList.pl command-line options	
Command-line option	**Description**
`-customer` *customer_name*	Optional; appends a text field to the output file.
`-debug` *debug_level*	Optional; specifies required debug level.
`-domain` *DomainName*	Mandatory; the name of the domain for which the discovery is running.
`-latency` *latency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-messageLevel` *messageLevel*	The level of messages to be logged (the default is warn): • debug • info • warn • error • fatal
`-outFile`	Optional; specifies the name of the file to write to.

discoAgentsUsed.pl

Use the **discoAgentsUsed.pl** script to determine which discovery agents were used to discover the most recently discovered devices in the current domain. Results are presented in an HTML file for viewing using a Web browser.

Description

This script produces a list of agents. The file containing this list is output to the following location:

```
current_directory/agentList.html
```

Note: The Discovery engine, ncp_disco, must be running when you run this script.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
discoAgentsUsed.pl -domain NCOMS
```

Command-line options

The following table describes the command-line options for the script.

Table 570. discoAgentsUsed.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory; the name of the domain where the discovery is running.
`-debug` *debug_level*	Optional; specifies required debug level.
`-latency` *latency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-messageLevel` *messageLevel*	Optional: The level of messages to be logged (the default is warn): • debug • info • warn • error • fatal

disco_profiling_data.pl

Use the **disco_profiling_data.pl** script to output summary data of all the discoveries run on a domain or extracted from a given profiling cache. This script includes data on how long it took to transfer discovery profiling data to the NCIM topology database. Data is sorted by discovery time.

Description

The script is run using the following command line. Optional arguments are shown enclosed in square brackets.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
disco_profiling_data.pl -domain domain_name  [ -fromcache ]
[ -discocachefile discovery_cache_filename ] [ -modelcachefile model_cache_filename ]
 [ -last ] [ -agents ] [ -debug debug_level ]
[ -help ]
```

The script reads data from the Topology manager database table, model.profilingData. For more information on this table see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Note: The Discovery engine, ncp_disco, must be running in order for this script to work.

Running the script

To retrieve data from a specified domain, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
disco_profiling_data.pl -domain NCOMS
```

To retrieve data from cache files, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
disco_profiling_data.pl -domain NCOMS -fromcache
```

To retrieve data from discovery and model caches, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
disco_profiling_data.pl -domain NCOMS
-discocachefile Disco.Cache.disco.profilingData.NCOMS
-modelcachefile Model.Cache.model.profilingData.NCOMS
```

Command-line options

The following table describes the command-line options for the script.

Table 571. disco_profiling_data.pl command-line options

Command-line option	Description
`-agents`	Returns profiling data on running discovery agents. Displays the following information: • Agent Name: The name of the agent. • Despatch: How many entities have been sent to the agent. • Interface Records: How many interface (component) records were in the responses. • Remote Neighbours: How many remote connection references were in the responses.
`-debug` *debug_level*	Optional; specifies required debug level.
`-discocachefile` *discovery_cache_filename*	Optional; name of a discovery cache file to extract disco profiling data from. This setting overrides the -fromcache setting.
`-domain` *DomainName*	Mandatory; the name of the domain to retrieve data from.
`-fromcache`	Optional; instructs the script to retrieve data from the cache files. In this case the Discovery engine, ncp_disco, and the Topology manager, ncp_model, do not need to be running.
`-help`	Optional; provides help on this command
`-last`	Optional. This integer specifies the last *n* discoveries for which to display statistics.
`-latest`	Optional: only the last discovery or ncp_model process data is displayed.
`-modelcachefile` *model_cache_filename*	Optional; name of a model cache file to extract model profiling data from. This setting overrides -fromcache setting.
`-stitchers`	Optional: displays a breakdown of how long each stitcher took to process the topology. Displays the following information: • StitcherName: The name of the stitcher. • Total: The total time in milliseconds that the stitcher took during the profiling interval, that is, from the start of phase 1 to the end of phase. • Executions: The number of times that the stitcher was run. • Average: The average time that the stitcher took, in milliseconds. • Total: The total percentage of the discovery processing time that the stitcher took.

Table 571. disco_profiling_data.pl command-line options (continued)	
Command-line option	**Description**
`-verbose`	Optional; provides more information on the screen.

Output

Running the script retrieves output similar to the following:

```
--------------------------------------------------------------------------------
Domain        Date_of_discovery     collection  processing   transfer     total
--------------------------------------------------------------------------------

NCOMS        2012-08-24T23:00:00    00:33:26    00:18:02    00:00:00    00:00:00
NCOMS        2012-09-30T23:00:04    00:30:36    00:16:04    00:11:04    00:57:44
NCOMS        2012-09-31T23:00:07    00:28:53    00:16:36    00:10:59    00:56:28
--------------------------------------------------------------------------------
```

```
-----------------------------------------------------------
entities  devices  access  interfaces  discoMem  modelMem
-----------------------------------------------------------

194328     352     347     93620        729.58    0.00
194925     352     348     93948        729.01    726.38
194997     352     348     93996        725.57    728.89
-----------------------------------------------------------
```

Table 572. Output columns	
Column	**Description**
Domain	Domain name.
Date_of_discovery	Start date and time of the discovery.
collection	Length of time spent collecting data. This is the sum of time spend in discovery phases 1-3.
processing	Length of time spent in the final processing phase of discovery.
transfer	Length of time taken for the Topology manager, ncp_model, to update NCIM following the discovery.
total	Total time taken for the discovery. This is the sum of collection, processing and transfer.
entities	Total number of entities discovered as reported by the Discovery engine, ncp_disco.
devices	Number of devices discovered as reported by the Discovery engine, ncp_disco.
access	Number of entities to which ncp_disco reported SNMP access interfaces.
interfaces	Number of interfaces discovered as reported by the Discovery engine, ncp_disco.
discoMem	Memory usage of the the ncp_disco process in MB.
modelMem	Memory usage of the ncp_model process in MB.

itnmMetaDiscoAudit.pl

Use the **itnmMetaDiscoAudit.pl** script to generate a report that contains audit information on device classification, and missing device metadata. The output also includes templates of SQL inserts that you can use to rectify missing metadata issues.

Generate a report

To run the script to generate a report, use a command similar to the following example:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnmMetaDiscoAudit.pl –domain NCOMS –report > my_report.txt
```

Generate a report for specific devices

To run the script to generate a report for specific devices use a command similar to the following example

Note: The -entity command-line option can be used multiple times, In this example it is used twice, once with an entity identifier, and the second time with an IP address.

:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnmMetaDiscoAudit.pl –domain NCOMS –report –entity 500 –entity 10.10.10.1
> my_report.txt
```

Generate a report to exclude specific device classes (AOCs)

To run the script to generate a report that excludes specific device classes, use a command similar to the following example:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnmMetaDiscoAudit.pl –domain NCOMS –report –exclude AIX –exclude Sun
```

View device membership for specified device classes (AOCs)

To run the script to view device membership for specified device classes, use a command similar to the following example:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnmMetaDiscoAudit.pl –domain NCOMS –showClassCisco –showClassIBM
```

Command line options

The following table describes the command line options for the script.

Table 573. itnmMetaDiscoAudit.pl command line options

Command line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain where you want to start, stop, or display status of a discovery.
`-report`	Optional; instructs the script to generate the output in a report.
`-showclass`	Optional; instructs the script to produce output that shows device membership for specified device classes. **Note:** The `-showclass` option cannot be used with the `-report` option.

Table 573. itnmMetaDiscoAudit.pl command line options (continued)	
Command line option	**Description**
`-version`	Prints the version and exits.
`-entity` *entity ID | entity name |IP address*	Optional; instructs the script to produce output for specific entities only. Entities can be specified by NCIM topology database entity identifier, by IP address, or by entity name. **Note:** This option can be used multiple times.
`-exclude` *parameter*	Optional; instructs the script to produce output that excludes specified parameters from the output tables; for example, entityId, className, or sysObjectId. **Note:** This option can be used multiple times.
`-maxTableRows` *number*	Limits the output table sizes to the specified number of rows. The default is 250 rows.
`-timeLimit` *seconds*	Limits the tool runtime to the specified number of seconds. The default is 300 seconds (5 minutes).
`-help`	Displays help.
`-debug`	Runs the script in debug mode.
`-v`	Optional; provides more information on the screen.

Script output

The script generates output in the following distinct sections:

AOC Class Hierarchy and Device Membership

Visualizes the AOC device class tree and indicates how many devices are in each class. The marker ### is used to bring specific AOC classes to your attention. For example, consider the following output snippet:

```
  1 Core
  5 |--- NetworkDevice 3 device(s) ###
127 | |--- Redback (Router)
 71 | |--- Dasan (Switch)
118 | |--- Nortel (NetworkDevice)
119 | | |--- BayWellfleet (Router)
120 | | |--- Centillion (Switch)
121 | | |--- NortelEthernetRoutingSwitch (Router)
123 | |  `-- NortelPassport (Switch)
124 | |   |--- NortelPassport15000 (Switch)
171 | |   |--- NortelPassport8xxx (Switch)
125 | |    `-- NortelPassport7000 (Switch)
358 | |--- Moxa (NetworkDevice)
359 | |  `-- MoxaNPortExpress (NetworkDevice)
220 | |--- RANBaseStation (Transmitter)
 11 | |--- Adtran (Router)
 12 | | |--- AdtranMX2800 (Router)
 13 | |  `-- AdtranNetVanta (Router)
```

The second line reads as follows:

```
  5 |--- NetworkDevice 3 device(s) ###
```

This line contains the following elements:

Device class identifier

In this line of output, the device class identifier is 5, corresponding to the `NetworkDevice` class.

Device class name
In this line of output, the device class name is `NetworkDevice`.

Number of devices in this class
This line of output reads `3 device(s)`. This statement means that the `NetworkDevice` class contains 3 devices.

Note: If there is no text after the class name, then there are no devices in the class. For example, in the preceding output snippet the only class that contains devices is the `NetworkDevice` class.

Suggestion to reclassify (*###*)
The presence of the *###* marker means that you should consider reclassifying the devices in the current device class into a more specific class. For example, in this line of output there are 3 devices in the `NetworkDevice` class. The `NetworkDevice` class is a container class and ideally would not directly contain devices: Container classes should contain other device classes rather than devices. If, for example, the three devices under `NetworkDevice` were a new type of Cisco device then these devices should be reclassified into a more specific class, such as `Cisco`, or, even better, into a subclass of `Cisco`.

For information on how to reclassify network devices, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Device Discovery Audit
Lists the devices with missing MIB data, and highlights the MIB data fields that are missing for each device listed. For each device, the following identifying information is provided:

- Entity identifier (entityId)
- Entity name
- IP address
- Device class
- MIB system object ID

For each device, the output lists MIB data fields and highlights missing fields with the marker *###*.

Metadata Audit
Lists the devices with missing metadata, and highlights the metadata fields that are missing for each device listed. For each device, the following identifying information is provided:

- Entity identifier (entityId)
- Entity name
- IP address
- Device class
- MIB system object ID

For each device, the output lists metadata fields and highlights missing fields with the marker *###*.

Recommended SQL Inserts
Lists recommended SQL insert templates. Use these templates to add the missing metadata to the database. The templates highlight the information that you need to add to get a working insert. For example, in the output that follows, you must provide values to replace the following dummy entries:

- `__deviceModel__`
- `__deviceFunction__`

```
INSERT INTO deviceFunction VALUES ('Avocent', '1.3.6.1.4.1.10418',
'1.3.6.1.4.1.10418.7.1.3', '__deviceModel__', '__deviceFunction__');
INSERT INTO deviceFunction VALUES ('Cisco', '1.3.6.1.4.1.9',
'1.3.6.1.4.1.9.1.1642', '__deviceModel__', '__deviceFunction__');
INSERT INTO deviceFunction VALUES ('Extreme Networks', '1.3.6.1.4.1.1916',
'1.3.6.1.4.1.1916.2.93', '__deviceModel__', '__deviceFunction__');
```

itnm_disco.pl

Use the **itnm_disco.pl** script to start and stop network discoveries, and display status of a running discovery.

Display the current status of network discovery

To run the script to display the current status of network discovery, use a command line similar to the following.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/itnm_disco.pl
-domain NCOMS -status -delay 5
```

Start a network discovery

To run the script to start a network discovery, use a command line similar to the following.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/itnm_disco.pl
-domain NCOMS -start
```

Note: If the **ncp_disco** process is running, then running the **itnm_disco.pl** script with the *-start* option starts a discovery. If the **ncp_disco** process is not running, then running the **itnm_disco.pl** script with the *-start* option only starts the **ncp_disco** process, and does not start a discovery.

Stop a network discovery

To run the script to stop a network discovery, use a command line similar to the following.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/itnm_disco.pl
-domain NCOMS -stop
```

Command-line options

The following table describes the command-line options for the script.

Table 574. itnm_disco.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain where you want to start, stop, or display status of a discovery.
`-status`	Optional; instructs the scripts to display status of a discovery.
`-start`	Optional; instructs the scripts to start a discovery.
`-stop`	Optional; instructs the scripts to stop a discovery.
`-delay`	Optional; count in seconds to delay
`-v`	Optional; provides more information on the screen.

listEntities.pl

Use the **listEntities.pl** script to retrieve device information from the ncimCache.entityData database table and output the information in HTML format.

Description

This script produces output to the following location:

```
current_directory/entityListing.html
```

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
listEntities.pl -domain NCOMS [ displayMode ] [ reportFileName ]
```

Command-line options

The following table describes the command-line options for the script.

Table 575. listEntities.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain where the discovery was run.
displayMode	Optional; a numerical value that indicates the level of detail to capture in the HTML file: • 0: Show just the types of device on the network • 1: Show each main node (each individual device) • 2: Show every entity from the ncimCache.entityData table including interfaces
reportFileName	Optional; name of the html file to generate. If not specified, defaults to `entityListing.html`.
`-debug` *debug_level*	Optional; specifies required debug level.
`-latency` *latency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-messageLevel` *messageLevel*	Optional: The level of messages to be logged (the default is warn): • debug • info • warn • error • fatal

restart_disco_process.pl

Use the **`restart_disco_process.pl`** script to stop the currently running discovery process and start a new instance. The running discovery process must have been started by ncp_ctrl for the script to take effect.

Description

The script stops the current discovery process by removing its entry from ncp_ctrl's services.inTray table. The script then inserts the entry into services.inTray again using the original argument list, triggering the process to restart. The `-startDiscovery` optional argument determines whether or not the script should wait for the discovery process to start and then initiate a new full discovery.

Running the script

The script is usually called from the RestartDiscoProcess discovery stitcher. However, you can run the script directly using the command line, for example:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
restart_disco_process.pl -domain NCOMS -startDiscovery 1
```

Command-line options

The following table describes the command-line options for the script.

Table 576. restart_disco_process.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory, the name of the domain on which to restart the discovery process.
`-debug 0-4`	Sets the debug level, where 0 is no logging and 4 is trace level logging.
`-help`	Displays command-line options help.
`-latency` *latency in seconds*	Time to wait for processing data in seconds.
`-startDiscovery 1|0`	Optional. If set to 1, then the script triggers a new full discovery after the new discovery process is running.

scheduleDiscovery.pl

Use the **`scheduleDiscovery.pl`** script to display when the next full discovery is scheduled. You can also use this script to schedule full discoveries.

Display the current discovery schedule

To display the current discovery schedule, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/
scheduleDiscovery.pl -domain NCOMS -display -v
```

Set a daily time for discovery

To set a daily time for discovery, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/
scheduleDiscovery.pl -domain NCOMS -time 24_hour_time -v
```

Set a weekly schedule for discovery

To set a weekly schedule for discovery, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/
scheduleDiscovery.pl -domain NCOMS -day 0..6 -time 24_hour_time -v
```

Set a monthly schedule for discovery

To set a monthly schedule for discovery, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/
scheduleDiscovery.pl -domain NCOMS -date 0..31 -time 24_hour_time -v
```

Set the discovery schedule to occur at a specified interval

To set the discovery schedule to occur at a specified interval, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/bin/
scheduleDiscovery.pl -domain NCOMS -interval number_of_hours_between_discovery -v
```

Command-line options

The following table describes the command-line options for the script.

Table 577. scheduleDiscovery.pl command-line options

Command-line option	Description
`-domain` *DomainName*	Mandatory; the name of the domain on which to schedule the discovery or query the discovery schedule.
`-time` *24_hour_time*	Optional; the time, in 24-hour clock format, at which to run to run discovery.
`-day` *day(s) of the week*	Optional; one or more days in a week to run discovery, where 0 is Sunday and 6 is Saturday.
`-date` *date(s) in the month*	Optional; one or more dates in the month when discovery must be run. If the date value is greater than 28, discovery might not execute in certain months.
`-interval` *number_of_hours_between_discovery*	Optional; number of hours between discovery.
`-v`	Turn on Verbose mode.

Example scripts

Use the example OQL and SNMP scripts as a starting point for creating your own scripts.

oql_example.pl

This script provides examples of Perl-scripted queries into the OQL databases. Use these examples as a starting point when writing your own script that uses the OQL extensions provided by ncp_perl.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
oql_example.pl
```

Command-line options

There are no command-line options for this script.

snmp_example.pl

This script provides examples of Perl-scripted SNMP queries into a specified device. Use this example script as a starting point when writing your own script that uses the SNMP extensions provided by ncp_perl.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_example.pl -node <Device>
```

Command-line options

The following table describes the command-line options for the script.

Table 578. snmp_example.pl command-line options

Command-line option	Description
-node *Device*	Mandatory; IP address or hostname for which the SNMP query.

Network Manager process management scripts

Use these scripts to query and control Network Manager processes.

create_all_control

On UNIX systems, use this script to configure processes to start automatically when your system is started or restarted, and to set up the **itnm_start**, **itnm_stop**, and **intm_status** scripts for your installation.

Running the script

Run this script on UNIX systems as the root user.

The following shows an example of using the script to configure your Network Manager processes to start automatically when your system is started or restarted.

```
$NCHOME/precision/install/scripts/create_all_control.sh -auto_only
```

Command-line options

The following table describes the command-line options for the **create_all_control** script.

Table 579. create_all_control command-line options

Command-line option	Description
-auto_only	Configures Network Manager processes to start automatically when your system is started or restarted.

Table 579. create_all_control command-line options (continued)	
Command-line option	**Description**
`-no_auto`	Sets up the **itnm_start**, **itnm_stop**, and **intm_status** scripts for your installation to make them available to be run manually when required. Does not configure the processes to start automatically when your system is started or restarted.
`-deinstall`	Removes the automatic start up for processes on system start or restart.
no option entered	If no command-line option is used, the script configures both the automatic start up and sets up the **itnm_start**, **itnm_stop**, and **intm_status** scripts for your installation to make them available to be run manually when required.

register_all_agents

During normal installation, the installation process should register all agents with the ncp_disco process. If for any reason this fails to happen, the script `register_all_agents` is provided so that the user can reregister the installed agent set. It should only be necessary to use this script on rare occasions.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/register_all_agents.sh
```

Command-line options

This script has no command-line options.

setup_run_as* scripts

Certain core processes in Network Manager must be run as root. If you installed Network Manager as a non-root user, then these processes will not be able to run unless you execute either the `setup_run_as_root.sh` or the `setup_run_as_setuid_root.sh` script on the server where the back-end processes are installed.

The `setup_run_as_root.sh` and `setup_run_as_setuid_root.sh` scripts affect which user can run the scripts that control the startup and shutdown of Network Manager as follows. The scripts affected are `itnm_start`, `itnm_stop`, and `itnm_status`.

Note: In order for the `setup_run_as_root.sh` and the `setup_run_as_setuid_root.sh` scripts to work correctly, you must be logged on as root when you run them.

Table 580. Who can run *itnm_start, itnm_stop, and itnm_status*		
Installing user	**Which setup_run_as* script was run**	**User who can run** **itnm_start** **itnm_stop** **itnm_status**
root user	Not applicable	root user only

Table 580. Who can run `itnm_start`, `itnm_stop`, and `itnm_status` (continued)		
Installing user	**Which setup_run_as* script was run**	**User who can run** **`itnm_start`** **`itnm_stop`** **`itnm_status`**
Non-root user	None	Installing user only
	`setup_run_as_root.sh`	root user only
	`setup_run_as_setuid_root.sh`	Users who are members of the same group as the installing user

setup_run_as_root.sh

If you installed Network Manager as a non-root user, and you want to run Network Manager while logged in as the root user, then you must run the `setup_run_as_root.sh` script to alter permissions so that you can run the back-end processes while logged on as the root user.

Note: In order for this script to work correctly, you must be logged on as root when you run it.

Running the script

To run the script, use a command line similar to the following.

Note: You must run this script before starting the Network Manager back-end processes.

```
$NCHOME/precision/scripts/setup_run_as_root.sh
```

Command-line options

This script has no command-line options.

setup_run_as_setuid_root.sh

If you installed Network Manager as a non-root user, and you want to run Network Manager while logged on as the non-root user who installed the product, or another user in the same group, then you must run the script `setup_run_as_setuid_root.sh`. The processes that must be run as root have their `setuid` permission changed so that they run as root even when started by a non-root user. This procedure has security implications and must not be done on a server that untrusted users can log in to.

Note: For this script to work correctly, you must be logged on as root when you run it.

Due to the way this script makes certain shared libraries into trusted libraries, only one installation per server can be set up to be run by a non-root user. If you have multiple installations of Network Manager on the same server, you must run all of them as root.

Running the script

To run the script, use a command line similar to the following example.

```
$NCHOME/precision/scripts/setup_run_as_setuid_root.sh
```

Command line options

This script has no command line options.

unsetup_run_as_setuid_root.sh

You can use this script to reverse the effects of the `setup_run_as_setuid_root.sh` script.

Note: In order for this script to work correctly, you must be logged on as root when you run it.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/scripts/unsetup_run_as_setuid_root.sh
```

Command-line options

This script has no command-line options.

setup_run_storm_as_non_root.sh

If you installed Network Manager as the root user, then you must run the `setup_run_storm_as_non_root.sh` script to enable the historical polling data processes to run. You do not need to run this script if you installed Network Manager as a non-root user.

Note: In order for this script to work correctly, you must be logged on as root when you run it.

Running the script

To run the script, use a command line similar to the following.

```
$NCHOME/precision/scripts/setup_run_storm_as_non_root.sh -g group_name
```

Command-line options

The following table describes the command line options for the `setup_run_storm_as_non_root.sh` script.

Table 581. setup_run_storm_as_non_root.sh command line options

Command-line option	Description
`-g group_name`	Mandatory; name of the poller aggregation group that specified when you installed the Network Manager core components. **Note:** When you installed the Network Manager core components, you specified values for both the poller aggregation group and the poller aggregation user, which is a member of the poller aggregation group. The poller aggregation user is listed in the `$NCHOME/precision/storm/conf/supervisor.conf` configuration file, within the `user` field of the `[supervisord]` section of that file. For more information, see the *IBM Tivoli Network Manager IP Edition Installation and Configuration Guide*.
`-h`	Displays help text for this script.

Polling scripts

Use these scripts to control and diagnose network polling,

get_policies.pl

Use the **get_policies.pl** Perl script to move poll policies and associated data between domains. This script can also be used to back up all poll policies to file or to load poll policies from a file into a specified domain. You can also move a subset of poll policies.

Description

Running the script

To run the script to copy policies from one domain to another, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
get_policies.pl -from domain=SOURCE -to domain=DESTINATION -ncim_password NCIM_password
 -ncmonitor_password NCMONITOR_password
```

To run the script to copy policies from one domain to a file, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
get_policies.pl -from domain=SOURCE -to file=exportedData.xml
```

Command-line options

The following table describes the command-line options for the script.

Table 582. `get_policies.pl` command-line options

Command-line option	Description
`-to` *domain=domainName*	Mandatory; This can be a named domain, in which case database connections are created to the NCIM and NCMONITOR databases configured for that domain.
`-to` *file=filename*	Mandatory; This can be a file name, in which case the contents of the file are expected to have been generated using this script, and have a format as outlined in the manual pages for NCP::Policies.
`-from` *domain=domainName*	Optional; This can be a named domain, in which case database connections are created to the NCIM and NCMONITOR databases configured for that domain. **Note:** If supplied, this considers the same format as `-to` *domain=domainName*. If not supplied, poll policies are taken from the default set, in `$NCHOME/precision/scripts/sql/monitorDefaultPolicies.xml`.
`-from` *file=filename*	Optional; This can be a file name, in which case the contents of the file are expected to have been generated using this script, and have a format as outlined in the manual pages for NCP::Policies. **Note:** If supplied, this considers the same format as `-to` *file=filename*. If not supplied, poll policies are taken from the default set, in `$NCHOME/precision/scripts/sql/monitorDefaultPolicies.xml`.

Table 582. `get_policies.pl` command-line options (continued)	
Command-line option	**Description**
`-password` *password*	Optional; the database password used for accessing the NCIM and NCMONITOR schemas. This is required only if the password is encrypted in the DbLogins configuration file.
`-policy` *policy name*	Optional; the policies that are copied and supplied to restrict the set of policies to write to the destination.
`-keep` *thresholds*|*policies*| *defaultPing*	Optional; this script is used to update a policy for one domain that will be passed on to subsequent domains.
`-viewTemplates` *filename*	Optional; this option is used to generate the network view information for policies with device scope defined, and write to a file conforming to Network Manager autoprovision template format.
`-triggerUpdates`	Optional; this option causes to set the policy.lastUpdate so that poller picks the changes to configuration if it is running.
`-commitPolicies` *yes*|*no*	Optional; when the destination is a database and if this option is set to yes, then it causes transactions to be committed with each policy written. If the option is set to no, then all policies are updated within a single transaction.
`-noViewTransfer`	Optional; this is used to prevent copying of any policyView data to the target system. This is desirable in cases where the source and target systems are not likely to share the same viewId data.
`-syncToBackup`	Optional; if a backup system is configured, updates it with changes to policies and templates on the primary, and delete any policies which are no longer exist on the primary.
`-help`	Optional; provides help on this command.

itnm_poller.pl

Use this script to enable and disable poll policies, check the status of poll policies and check the polling status of IP addresses.

You can use the script to monitor the health of **`ncp_poller`** processes.

- "Syntax" on page 932
- "Examples" on page 935

Syntax

Run the script with the following syntax:

```
ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain domain [-poller pollername]
[-enable policyid|-disable policyid] [-status all|static|realtime]
[-refresh policyid|all] [-chassis ipaddress|filename][-interface ipaddress|filename] [-metrics]
[-window] [-timestamp timestamp] [-help]
```

The following table describes the options for the script. To obtain poll policy IDs for use with these options, run the script with the `-status` option first.

Table 583. `itnm_poller.pl` options	
Command-line option	**Description**
`-domain` *domainName*	Required: The domain that contains the poll policies of interest.

Table 583. `itnm_poller.pl` options (continued)	
Command-line option	**Description**
`-chassis` *ipaddress*|*filename*	Optional: Displays the polling status of a single chassis ping poller if a specified IP address is supplied as the argument, or displays the status of multiple chassis ping pollers if the name of a text file that contains a list of IP addresses is supplied as the argument. See also; -monitors.
`-disable` *policyid*	Optional: Disables the poll policy that has the specified poll policy ID
`-enable` *policyid*	Optional: Enables the poll policy that has the specified poll policy ID.
`-help`	Optional: Displays help text.
`-interface` *ipaddress*|*filename*	Optional: Displays the polling status of a single interface ping poller if a specified IP address is supplied as the argument, or displays the status of multiple interface ping pollers if the name of a text file that contains a list of IP addresses is supplied as the argument. See also; -monitors.
`-metrics`	Reads the metrics trace and displays the information on the command-line interface. The script first looks for the metrics trace file in the current working directory. If the file is not found, it looks in `$NCHOME/precision/logs`. The information is displayed as a bar chart for each metric. The chart plots the most recent data in the trace file, showing the past 4 hours of data by default. For each metric, the end time is determined by the time stamp of the latest entry for the metric in the trace. Alternatively, the end time can be set by using the `-timestamp` option. There are 5 metrics. For the first metric, which is called Health, a separate bar chart is displayed for each combination of poll policy and poll definition that is associated with the poller. To ensure that the bar charts display correctly, run the script with this option on a terminal that is no narrower than 140 characters. **Important:** Do not use this option with the `-status` option. If you do, the script displays a message and terminates.
`-monitors` *ipaddress*|*filename*	Optional: Displays the polling status of a single ping poller (either chassis or interface ping poller) if a specified IP address is supplied as the argument, or displays the status of multiple ping pollers if the name of a text file that contains a list of IP addresses is supplied as the argument.

Table 583. `itnm_poller.pl` options (continued)

Command-line option	Description
`-poller` *pollername*	Optional: For use in environments that use multiple pollers. Specify the identifier of the poller for which you want to output information. If you omit the `-poller` option, information for the default poller is displayed. To explicitly specify the default poller, use the value ncp_poller_default. For example: `$NCHOME/precision/bin/ncp_perl itnm_poller.pl -domain NCOMS -status static -poller ncp_poller_default` **Note:** The `-poller` option is no longer just for use in conjunction with the `-metrics` option. From V4.2 onwards, the `-poller` option is applicable to all options. If no `-poller` is specified, then the output retrieved is as follows: • If you specify the `-metrics` option, then the command retrieves metrics for the default poller only. • If you specify one of the other options, then the command retrieves information for all of the pollers.
`-refresh` *policyid*|all	Optional: Refreshes the policy configuration and its entity list. To refresh a single policy, specify the policy ID. To refresh all policies, use the `-refresh all` option.
`-status` all|static|realtime	Optional: Displays the status of the specified policies, and also the ID of each poll policy. Possible options are as follows: • `all` • `static` (default) • `realtime` **Important:** Do not use this option with the `-metrics` option. If you do, the script displays a message and terminates.
`-timestamp` *timestamp*	Optional: For use with the`-metrics` option. Specify the time stamp for the end time, from which to read the metrics data. This option overrides the default end time of the last time stamp in the trace for each metric. Use this option together with the `-window` option to obtain metrics for specific periods of time. The `-timestamp` option can also be used without the `-window` option to obtain the default time period of the last 4 hours.

Table 583. `itnm_poller.pl` options (continued)	
Command-line option	**Description**
`-window`	Optional: For use with the `-metrics` option. Specify the time period, in multiples of 4 hours, for which you want to display metrics data for the poller. The value of this option affects the x-axis of the bar charts. The default is 4. The time period is calculated from a specified time stamp or from the most recent time stamp in the trace file. If you do not specify a multiple of 4, the time period is rounded up to the nearest 4 hours. The most useful time period is 4-24 hours. Optionally use this option together with the `-timestamp` option to obtain metrics for specific periods of time.

Examples

- "Display poll policy status and ID" on page 935
- "Enable poll policies" on page 935
- "Disable poll policies" on page 935
- "Trigger the refresh of a policy configuration and its entity list" on page 936
- "Display polling status of a chassis ping poller" on page 936
- "Display poller health charts" on page 936

Display poll policy status and ID

The following example displays the status of all poll policies on the NCOMS domain. It also displays the policy IDs of all poll policies, so it is useful to identify poll policies for subsequent actions, for example, enabling or refreshing a policy.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -status all
```

The following example displays the status of the poll policies for the default poller, ncp_poller_default, on the NCOMS domain:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -status all -poller ncp_poller_default
```

Enable poll policies

The following example enables a poll policy that has the ID 10 on the NCOMS domain.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -enable 10
```

Disable poll policies

The following example disables a poll policy that has the ID 10 on the NCOMS domain.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -disable 10
```

Trigger the refresh of a policy configuration and its entity list

The following example triggers a refresh of the policy configuration for a poll policy that has the ID 10 on the NCOMS domain.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -refresh 10
```

Display polling status of a chassis ping poller

The following example displays the polling status of the chassis ping poller that has the IP address 10.101.10.10 on the NCOMS domain.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
itnm_poller.pl -domain NCOMS -chassis 10.101.10.10
```

Display poller health charts

The following example outputs the bar charts with the default settings. The metrics are output for the poll policies and poll definitions that are associated with the default poller. The time period for the metrics is 4 hours before the most recent set of metrics that were written to the trace, so 4 hours of data are recorded on the x-axis.

```
$NCHOME/precision/bin/ncp_perl itnm_poller.pl -domain NCOMS -metrics
```

The following example outputs the bar charts for a poller that is called 2345_POLL. The time period for the metrics starts 24 hours before the most recent time stamp.

```
$NCHOME/precision/bin/ncp_perl itnm_poller.pl -domain NCOMS -metrics
 -poller 2345_POLL -window 24
```

The following example outputs the bar charts for the default poller. The time period for the metrics starts from the time stamp that was 4 hours before 09:14 and 59 seconds on December 10, 2013

```
$NCHOME/precision/bin/ncp_perl itnm_poller.pl -domain NCOMS -metrics
-timestamp 2013-12-10T09:14:59
```

The following example outputs the bar charts for the poll policies and poll definitions for a poller that is called 2345_POLL. The time period for the metrics starts from the time stamp that was 8 hours before 09:14 and 59 seconds on December 10, 2013.

```
$NCHOME/precision/bin/ncp_perl itnm_poller.pl -domain NCOMS -poller 2345_POLL
-metrics -window 8 -timestamp 2013-12-10T09:14:59
```

For more information about the NCMONITOR polling status tables, including the ncmonitor.expectedIps table, see the *IBM Tivoli Network Manager Reference*. For more information about how to ensure that the important IP addresses in your network are polled as expected, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

ncp_ping_poller_snapshot.pl

This script is used for troubleshooting ping polling of network devices. After the ncp_upload_expected_ips.pl script has uploaded a plain text file of IP addresses, the ncp_ping_poller_snapshot.pl script creates and stores a snapshot of the current ping polling status of these addresses. You can then run a report on these devices using the ncp_polling_exceptions.pl script.

Description

The ncp_ping_poller_snapshot.pl script retrieves the polling status of the uploaded IP addresses; that is, whether they will be polled by the ncp_poller process. The polling status of devices can change after a network discovery or a change in polling configuration.

The data retrieved by this script is stored in the pollLog database table in the NCMONITOR schema, and can be used to generate reports on the polling status using the ncp_polling_exceptions.pl script.

For more information on ensuring that the important IP addresses in your network are being polled as expected by Network Manager, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Prerequisites for this script are as follows:

- You must have run the ncp_upload_expected_ips.pl script on a valid file of IP addresses.
- The pollLog and pollLogSummary tables must have been created in the NCMONITOR schema.
- The DbLogins file must be usable for the given domain.
- The domain must exist in the NCIM topology database.
- The Polling engine, **ncp_poller**, must be running in the given domain.
- There must be at least one active ping poll in the current domain.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_ping_poller_snapshot.pl -domain DOMAIN_NAME -password PASSWORD
```

Command-line options

The following table describes the command-line options for the script.

Table 584. ncp_ping_poller_snapshot.pl command-line options

Command-line option	Description
`-domain` *DOMAIN_NAME*	Mandatory; the name of the relevant domain.
`-password` *PASSWORD*	Optional; the database password used to access the NCIM and NCMONITOR schemas. This is required only if the password is encrypted in the DbLogins configuration file.
`-logdir`*LOGDIRNAME*	Optional. The directory in which the log file is generated. A log file called `ncp_ping_poller_snapshot.pl.`*DOMAIN_NAME*`.log` can be checked if there are any problems accessing the database. It is generated in the current directory by default if this option is not given.
`-help`	Optional; provides help on this command

ncp_polling_exceptions.pl

This script is used for troubleshooting ping polling of network devices. After having run the ncp_upload_expected_ips.pl and ncp_pingpoller_snapshot.pl scripts, use this script to print a report of polling status of network devices.

Description

After uploading a list of IP addresses that you want to monitor using the ncp_upload_expected_ips.pl script, and creating a snapshot of the polling status of those devices using the ncp_pingpoller_snapshot.pl script, use this script to print a report of the snapshot data. The script lists those addresses that are not being polled using ICMP, and indications why they are not being polled.

For more information on the procedure to ensure that the important IP addresses in your network are being polled as expected by Network Manager, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_polling_exceptions.pl -domain DOMAIN_NAME -format list | report
```

Command-line options

The following table describes the command-line options for the script.

Table 585. ncp_polling_exceptions.pl command-line options

Command-line option	Description
`-domain` *DOMAIN_NAME*	Mandatory; the name of the relevant domain.
`-notpolled`	Optional; outputs a list of IP addresses that are not polled as compared with the list of expected IP addresses. This output is in LIST format only.
`-format` list | report	Optional; determines the output format. This can be report format or a list of IP addresses. If omitted, defaults to report output.
`-help`	Optional; provides help on this command

ncp_upload_expected_ips.pl

This script is used for troubleshooting ping polling of network devices. Use the ncp_upload_expected_ips.pl script to upload a plain text file of IP addresses. Use the ncp_pingpoller_snapshot.pl and ncp_pollingexceptions.pl scripts to check the uploaded addresses.

Description

Use the ncp_upload_expected_ips.pl script as part of the procedure to ensure that the important IP addresses in your network are being ping polled as expected by Network Manager and, if not, to provide information to resolve the problem.

The script loads a list of IP addresses to the ncmonitor.expectedIps table. Any data already in the table is removed.

Run the ncp_upload_expected_ips.pl script before running the ncp_pingpoller_snapshot.pl and ncp_pollingexceptions.pl scripts. You can run the ncp_pingpoller_snapshot.pl and ncp_pollingexceptions.pl scripts many times after running the ncp_upload_expected_ips.pl script once. Run the ncp_upload_expected_ips.pl again when the IP addresses that you want to check have changed.

For more information on the NCMONITOR polling status tables, including the ncmonitor.expectedIps table, see the *IBM Tivoli Network Manager Reference*.

For more information on the procedure to ensure that the important IP addresses in your network are being polled as expected by Network Manager, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Prerequisites for this script are as follows:

- A plain text file containing IP addresses you want to monitor is available on the local file system.
- The expectedIps table must have been created in the NCMONITOR schema.
- The DbLogins file must be usable for the given domain.
- The domain must exist in the NCIM topology database.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_upload_expected_ips.pl -domain DOMAIN_NAME -file FILENAME -password PASSWORD
```

Command-line options

The following table describes the command-line options for the script.

Table 586. ncp_upload_expected_ips.pl command-line options

Command-line option	Description
`-domain` *DOMAIN_NAME*	Mandatory; the domain that contains the IP addresses for which you want to check polling status.
`-file` *FILENAME*	Mandatory; a plain text file of IP addresses, separated by whitespace (for example, one IP address per line). The script accepts IPv4 addresses only. The file is expected to contain just IP addresses in standard dot notation.
`-password` *PASSWORD*	Optional; the database password used to access the NCIM and NCMONITOR schemas. This is required only if the password is encrypted in the DbLogins configuration file.
`-logdir`*LOGFILENAME*	Optional; a log file called ncp_upload_expected_ips.*DOMAIN_NAME*.log is generated that can be checked if there are any problems accessing the database. It is generated in the current directory by default if this option is not given.
`-help`	Optional; provides help on this command

SQL scripts

Use the supplied SQL scripts to perform setup tasks on the Tivoli Netcool/OMNIbus ObjectServer.

create_itnm_triggers.sql

Use this script to set up a Tivoli Netcool/OMNIbus ObjectServer to support the setting of event severity based on the value of the NmosCauseType field. For example, if NmosCauseType has the value 1 (root cause), then running this script will cause the event severity to be set to `Critical`.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/omnibus/bin/nco_sql -server objectserver_name -user user_name -password
 password < $NCHOME/precison/scripts/create_itnm_triggers.sql
```

Command-line options

The following table describes the command-line options for the **`create_itnm_triggers.sql`** script.

Table 587. create_itnm_triggers.sql command-line options	
Command-line option	**Description**
objectserver_name	Mandatory; the name of the database.
user_name	Mandatory; the name of the database user.
password	Mandatory; the password of the database user.

create_sae_automation.sql

Use this script to set up the Tivoli Netcool/OMNIbus ObjectServer with automations and right-click tools to support the generation of service-affected events.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/omnibus/bin/nco_sql -server objectserver_name -user user_name
-password password < $NCHOME/precision/scripts/create_sae_automation.sql
```

Command-line options

The following table describes the command-line options for the **create_sae_automation.sql** script.

Table 588. create_sae_automation.sql command-line options	
Command-line option	**Description**
objectserver_name	Mandatory; the name of the database.
user_name	Mandatory; the name of the database user.
password	Mandatory; the password of the database user.

drop_itnm_triggers.sql

Use this script to set up a Tivoli Netcool/OMNIbus ObjectServer to remove support for setting of event severity based on the value of the NmosCauseType field. After running this script, the value of NmosCauseType (for example 1,root cause) has no effect on event severity.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/omnibus/bin/nco_sql -server objectserver_name -user user_name -password
 password < $NCHOME/precison/scripts/drop_itnm_triggers.sql
```

Command-line options

The following table describes the command-line options for the **drop_itnm_triggers.sql** script.

Table 589. drop_itnm_triggers.sql command-line options	
Command-line option	**Description**
objectserver_name	Mandatory; the name of the database.
user_name	Mandatory; the name of the database user.
password	Mandatory; the password of the database user.

drop_sae_automation.sql

Use this script to remove from the Tivoli Netcool/OMNIbus ObjectServer the automations and right-click tools to support the generation of service-affected events.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/omnibus/bin/nco_sql -server objectserver_name -user user_name
-password password < $NCHOME/precision/scripts/drop_sae_automation.sql
```

Command-line options

The following table describes the command-line options for the **drop_sae_automation.sql** script.

Table 590. drop_sae_automation.sql command-line options

Command-line option	Description
objectserver_name	Mandatory; the name of the database.
user_name	Mandatory; the name of the database user.
password	Mandatory; the password of the database user.

Troubleshooting scripts

Use these scripts to perform troubleshooting tasks.

GetDiscoCache.pl

To generate discovery cache files for a recent discovery as if it had been run in failover mode, use the GetDiscoCache.pl Perl script. Failover cache files help IBM Support and Development teams to troubleshoot discovery.

Running the script

After a discovery has finished, you can run the GetDiscoCache.pl script to generate cache files for that discovery. The ncp_disco process must still be running. If the ncp_disco process has been stopped or restarted since the discovery finished, you cannot use the GetDiscoCache.pl script to generate cache files for that discovery. You must either run another discovery in failover mode or run another discovery normally and then run the GetDiscoCache.pl script.

The script stores the cache files in `ITNMHOME/var/precision` as `PerlStore.`*`timestamp`*`.Cache.`*`DatabaseName`*`.`*`TableName`*`.`*`DomainName`*, so that they are ready to send to IBM Support for troubleshooting purposes. To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
GetDiscoCache.pl -domain DomainName
```

Note: On UNIX systems only, the script also compresses the cache files into a `.tar` file by default. For more information, see option `-buildtar` in the following table.

Command-line options

The following table describes the command-line options for the **GetDiscoCache.pl** script.

Table 591. GetDiscoCache.pl command-line options	
Command-line option	**Description**
`-domain` *DomainName*	Mandatory; the name of the domain you want to retrieve discovery tables for and build a copy of the cache files.
`-buildtar` *0* | *1*	On UNIX systems only; sets whether the copy of the cache files is compressed into a `.tar` file in the current directory. The file is called *service*.*timestamp*.*DomainName*.tar, where *service* is the name of the process from which to retrieve the data. The default setting is `1`, meaning the files are compressed. Set it to `0` to turn off the creation of compressed `.tar` files.
`-dbName`	Optional; specifies the discovery database to cache (for example, `Details`).
`-debug` *DebugLevel*	Optional; the level of detail the debugging output provides. Values are 1 to 4, where 4 represents the most detailed output.
`-entityNames`	Optional; filter cache of selected devices.
`-largeTables`	Optional; list of database tables that must be processed in smaller chunks. Only use when the `-service` option is not set, or when the `-service` option is set to `Disco`.
`-latency` *MessageLatency*	Optional; the maximum time in milliseconds to wait between attempts to send a message. This is needed for busy networks.
`-service`	Optional. You can specify the name of the service to retrieve the cache data from. The default is `Disco` for the discovery process. You cannot use the `Objectserver` or `Ncim` services.
`-tblName`	Optional; specifies the discovery table to cache. Only used if `-dbName` is set (for example, `-dbName IpRoutingTable -tblName returns`).
`-help`	Optional; displays help for command line options on screen.

The following is an example of using the command-line options for **`GetDiscoCache.pl`** on a UNIX system:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
GetDiscoCache.pl -domain NCOMS -service Disco -buildtar 1
```

ncp_db_access.pl

Checks database setup and determines whether access to the databases is being prevented by firewalls. This script accesses the topology database, historical polling database, and the distributed polling database.

Running the script

The script uses the domain-specific `DbLogins.`*`DOMAIN`*`.cfg` and `MibDbLogin.`*`DOMAIN`*`.cfg` files for access credentials. If there are no domain-specific versions of these files, then the script uses the standard `DbLogins.cfg` and `MibDbLogin.cfg` files.

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_db_access.pl -domain NCOMS
```

Command-line options

The following table describes the command-line options for the script.

Table 592. ncp_db_access.pl command-line options

Command-line option	Description
`-domain` *Domain*	Mandatory; the domain in which to check database access.
`-help`	Prints help text.
`-verbose`	Prints connection details as the script tries to connect. **Note:** Script default is to simply indicate whether the connection was successful.

ncp_storm_validate.sh

Use this script as a troubleshooting aid for the Apache Storm realtime computation system, which is used to aggregate raw poll data into historical poll data.

Syntax

The `ncp_storm_validate` script uses the following syntax:

```
$NCHOME/precision/scripts/ncp_storm_validate.sh [storm] testName testArgs
```

Where:

- `storm` is an optional argument used to additionally use Storm scripts to trigger Java code. By default, the Java code is triggered directly to reduce unhelpful messages. Add the optional storm argument to run this script using the Storm scripts.
- *testName* is the name of the test to run.
- *testArgs* are the arguments required by that test.

Examples

Here are some examples of how to run the script:

Display the current default configuration

```
$NCHOME/precision/scripts/ncp_storm_validate.sh config
```

Display the configuration for the named topology, "NMAnotherTopology"

```
$NCHOME/precision/scripts/ncp_storm_validate.sh config NMAnotherTopology
```

Validate access to the NCPOLLDATA database using existing credentials

```
$NCHOME/precision/scripts/ncp_storm_validate.sh db
```

Validate access to the NCPOLLDATA database using existing credentials but by additionally using the Apache Storm scripts

```
$NCHOME/precision/scripts/ncp_storm_validate.sh storm db
```

Fix Pack 1 **Delete all historical poll data aggregated by Storm**

```
$NCHOME/precision/scripts/ncp_storm_validate.sh clear -aggregate
```

Command-line options

The following table describes the command-line options for the **ncp_storm_validate** script. In all cases, [*topology_name*] defaults to NMStormTopology if not provided.

Table 593. ncp_storm_validate.sh command-line options

Command-line option	Description
Fix Pack 1 Fix Pack 1 `clear [topology_name] [-raw | -aggregate | -all]`	Deletes historical poll data. You will be prompted to confirm this action before any data is deleted. You can delete the following historical poll data: • `-raw`: deletes the raw poll data stored by the poller, and as yet unprocessed by Storm. In practice, this is the last hour's data from the pollData and pollBatch tables within the NCPOLLDATA database. • `-aggregate` - deletes the historical poll data aggregated by Storm. This is data older than an hour and includes all of the data in all of the historical poll data tables in the NCPOLLDATA database. These are the tables with the prefix pdEwmaFor; for example pdEwmaForDay. • `-all` - deletes both raw and aggregated data. **Warning:** This option is intended for use during setup and testing only, and is not for use during production use.
`config [topology_name]`	Displays the specified Storm topology. If you do not specify a topology name, then the command displays the default topology NMStormTopology.
`crypt [topology_name] -password password [-decrypt]`	Encrypts or decrypts a password from logs. Displays the plain text decrypted password. **Note:** This option is not compatible with the DbLogins.cfg configuration file. Use the ncp_crypt command for working with that file.

Table 593. ncp_storm_validate.sh command-line options (continued)

Command-line option	Description
`db [topology_name]`	Validates access to the NCPOLLDATA database. Tries to connect to the database using the existing credentials. Running this script with this option produces similar output to the ncp_db_access.pl script, but also validates the JDBC connection.
`dbconfig [topology_name]`	Displays NCPOLLDATA database configuration. Identifies the JDBC URLand username used to access the database. This information is defined by the DbLogins.cfg configuration file for the named domain and is potentially overridden by the jdbc.url value from the backend tnm.properties configuration file.
`dblogins [topology_name]`	Loads the backend DbLogins.cfg file using JNI. JNI loading of the DbLogins configuration is a two-step process. 1. The path to the C++ library (libNcpDbJni.so) is given in the java.library.path property of the `$NCHOME/precision/storm/conf/storm.yaml` file. That path value should never need to be modified. 2. To load up the library, the library path (`LD_LIBRARY_PATH` on linux, `LIBPATH` on AIX) must be set to pick up further C++ dependencies. This should happen by default based on settings picked up automatically from `$NCHOME/precision/bin/ncp_common`.
`hb [topology_name]`	Displays the current status of the Apache Storm master table in the NCPOLLDATA database. Use this option to identify the master topology, and show the current timestamp. By running the script with this option over a number of minutes, you are able to display an incrementing batch identifier, and this indicates that the poller is continuously storing data.
`hbconfig [topology_name]`	Displays heartbeat configuration for the Apache Storm master table in the NCPOLLDATA database.
Fix Pack 1 Fix Pack 1 `kafkaexport`	Exports sample data for selected topics. The topic selected determines the output format. For a full description of the parameters for this option see “Exporting sample data using the kafkaexport option” on page 946.
Fix Pack 1 Fix Pack 1 `kafkaimport`	Listens for data requests on a specified kafka topic or for a specified number of seconds. For a full description of the parameters for this option see “Listening for data requests using the kafkaimport option” on page 947.
`keyfile [keyFileName]`	Validates an existing encryption key or creates a new one if no key exists.

Exporting sample data using the kafkaexport option

Fix Pack 1

The kafkaexport option uses the following syntax:

```
kafkaexport [topology_name] -topic topic [-clientid clientid]
[-instanceid monitoredInstanceId] [-objectid monitoredObjectId]
[-value value] [-message messageString]
```

Here are some examples of how to export sample data using the kafkaexport option.

Note: The topic names are case sensitive. In normal use you do not need to be concerned with topic names. A scenario in which you do need to take care, however, is when troubleshooting with the ncp_storm_validate.sh script. In this case be aware that you must type the topic name exactly as spelled here; that is, all lowercase, for example: nm.monitoredobject.

Fix Pack 1 Export test data to kafka

The format is hard coded based on the topics configured. The following examples show relevant options for different topic configurations.

```
ncp_storm_validate.sh kafkaexport -topic nm.polldata
```

```
ncp_storm_validate.sh kafkaexport -topic nm.polldata -value 10
-instanceid 4 -objectid 3
```

```
ncp_storm_validate.sh kafkaexport -topic nm.monitoredinstance
```

```
ncp_storm_validate.sh kafkaexport -topic nm.monitoredinstance -instanceid 4
```

```
ncp_storm_validate.sh kafkaexport -topic nm.monitoredobject
```

```
ncp_storm_validate.sh kafkaexport -topic nm.monitoredobject -objectid 3
```

Fix Pack 1 Request a full table dump

The dump option requires Apache Storm to be running. The following examples show relevant options for different topic configurations.

```
ncp_storm_validate.sh kafkaexport -topic nm.datarequest
-message monitoredobject
```

```
ncp_storm_validate.sh kafkaexport -topic nm.datarequest
-message monitoredinstance
```

Table 594. Parameters for the kafkaexport option

Command-line option	Description
topology_name	Exports sample data from the specified Storm topology. If you do not specify a topology name, then the command exports sample data from the default topology NMStormTopology.
-topic *topic*	The topic for which you want to export poll data. Options are: • nm.datarequest • nm.monitoredinstance • nm.monitoredobject • nm.polldata

Table 594. Parameters for the kafkaexport option (continued)	
Command-line option	**Description**
`-clientid` *clientid*	This identifier is autogenerated by the system and is logged in the relevant Apache Storm log file. You do not normally need to specify this value. **Note:** If you need to troubleshoot issues, then you can specify a value for `clientid` by modifying the value logged in the the relevant Apache Storm log file.
`-instanceid` *monitoredInstanceId*	Dummy value that can be specified if an nm.polldata or nm.monitoredinstance topic was specified. The value specified here will be exported. **Note:** This value is not used to look up a row in the NCIM topology database.
`-objectid` *monitoredObjectId*	Dummy value that can be specified if an nm.monitoredobject topic was specified. **Note:** This value is not used to look up a row in the NCIM topology database.
`-value` *value*	Can be used with the topic nm.polldata to specify a dummy value for test purposes.
`-message` *messageString*	Can be used with the topic nm.datarequest to identify the type of data requested. It currently must be either 'monitoredobject' or 'monitoredinstance', triggering a full dump of the named table in each case.

Listening for data requests using the `kafkaimport` option

Fix Pack 1

The `kafkaimport` option uses the following syntax:

```
kafkaimport [topology_name] -topic topic [-runseconds runseconds] [-groupid groupId]
```

Here are some examples of how to listen for data requests using the `kafkaimport` option. The format of the output depends on the topics configured.

Note: The topic names are case sensitive. In normal use you do not need to be concerned with topic names. A scenario in which you do need to take care, however, is when troubleshooting with the `ncp_storm_validate.sh` script. In this case be aware that you must type the topic name exactly as spelled here; that is, all lowercase, for example: `nm.monitoredobject`.

```
ncp_storm_validate.sh kafkaimport -topic nm.monitoredinstance
```

```
ncp_storm_validate.sh kafkaimport -topic nm.monitoredobject
-runseconds 100000
```

```
ncp_storm_validate.sh kafkaimport -topic nm.polldata
```

Table 595. Parameters for the kafkaimport option	
Command-line option	**Description**
topology_name	Listens for data requests from the specified Storm topology. If you do not specify a topology name, then the command listens for data requests on the default topology NMStormTopology.
-topic *topic*	The topic for which you want to listen for data requests. Options are: • nm.datarequest • nm.monitoredinstance • nm.monitoredobject • nm.polldata
-runseconds *runseconds*	The number of seconds for which you want to listen for data requests.
-groupid *groupId*	This identifier is autogenerated by the system and is logged in the relevant Apache Storm log file. You do not normally need to specify this value. **Note:** If you need to troubleshoot issues, then you can specify a value for `groupid` by modifying the value logged in the the relevant Apache Storm log file.

ncp_validate_ncim_tables.pl

This script compares the created NCIM tables and views with the tables and views defined in the schema files, to verify that all defined tables and views have been created. Run this script if you suspect that there are issues with the NCIM database, for example after migration.

Description

Before running this script, you must ensure that NCIM database credentials are available in a `$NCHOME/etc/precision/DbLogins.cfg` file for the domain that you want to check.

Running the script

To run the script, use a command line similar to one of the following commands:

Running the script silently and print failures

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_validate_ncim_tables.pl -domain TEST
```

Printing all created tables and failures

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_validate_ncim_tables.pl -domain TEST -verbose
```

Command-line options

The following table describes the command-line options for the script.

Table 596. ncp_validate_ncim_tables.pl command-line options	
Command-line option	**Description**
`-domain` *Domain*	Mandatory; the domain that you want to check.
`-help`	Optional; provides help on this command
`-verbose`	Optional; displays progress to stdout. Prints a list of all successfully created database tables as well as tables that could not be created. If this option is not specified, only failures are logged.

PrintCacheFile.pl

Takes a specified cache file and prints the ROMP contents of the file in a human-readable format. This script is mostly useful for scripting and debugging purposes.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
PrintCacheFile.pl -domain domain cache_file
```

Command-line options

The following table describes the command-line options for the script.

Table 597. PrintCacheFile.pl command-line options	
Command-line option	**Description**
`-domain` *domain*	The domain in which you want the script to run. This option is mandatory.
`cache_file`	The name of the cache file, including the path to the file. This option is mandatory.
`-restore` *printedCacheFileName*	Fix Pack 3 Restores a previously printed cache file *printedCacheFileName* into the cache file specified by the **`cache_file`** option.

snmp_walk.pl

Performs an SNMP walk of all or part of a device using the SNMP helper.

Description

The **`snmp_walk.pl`** script walks the entire device MIB (starting from `internet` by default) or part of the MIB.

This script accesses devices through the SNMP Helper. You can configure device access using the **Discovery Configuration GUI**.

By default, the script writes output to screen. You can change how the script writes output using the `-format` option.

Restriction:

Before running this script, ensure that the Helper Server process, **ncp_d_helpserv**, and either the master process controller, **ncp_ctrl**, or the SNMP helper process, **ncp_dh_snmp**, are running in the required domain.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain domain_name [
 -debug ] [ -format ] [ -help ] [ -ignoreInstanceFilters ] [ -latency ] [ -quiet ]
 IP_address [ OID|MIB_variable [ community_add_on ] ]
```

Note: You can configure this script to ignore SNMP interface filters. For more information on SNMP interface filtering, see the *IBM Tivoli Network Manager IP Edition Administration Guide*.

Examples

The following example command line performs a complete walk of a device with IP address 1.2.3.4. Output is written to screen.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain TEST 1.2.3.4
```

The following example command line performs a complete walk of a device with IP address 1.2.3.4. Output is written to screen and to a file.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain TEST -format out -format name 1.2.3.4
```

The following example command line performs a walk of the ifTable of a device with IP address 1.2.3.4. The ifTable is specified as a MIB variable. Output is written to screen.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain TEST 1.2.3.4 ifTable
```

The following example command line performs a walk of the ifTable of a device with IP address 1.2.3.4. The ifTable is specified as an OID. Output is written to screen.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain TEST 1.2.3.4 1.3.6.1.2.1.2.2
```

The following example command line performs a walk of the ifTable of a device with IP address 1.2.3.4 with a VLAN context. Output is written to screen.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
snmp_walk.pl -domain TEST -format out 1.2.3.4 ifTable @vlan2
```

Command-line options

The following table describes the command-line options for the script.

Table 598. snmp_walk.pl command-line options

Command-line option	Description
-debug	Optional; sets the debug level for the process, from 1 to 4.
-domain *Domain*	Mandatory; the domain containing the device on which to perform the SNMP walk.

Table 598. snmp_walk.pl command-line options (continued)	
Command-line option	**Description**
`-format`	By default, the script writes output to screen. You can change this behavior using the -format option. Possible options are: • `name` - writes the retrieved data, by name, to file. The file is called *`IP_address.Starting_OID.MIB_name`*. • `oid` - writes the retrieved data, by OID, to file. The file is called *`IP_address.Starting_OID.OID_name`*. • `out` - writes the retrieved data, by name, to screen (stdout). You can specify more than one *-format* option.
`-help`	Optional; provides help on this script.
`-ignoreInstanceFilters`	Optional; returns all information from the device, ignoring any interface filters.
IP_address	An IP address of the device. The address must be accessible using SNMP. You cannot use device names.
`-latency`	Optional; specifies the timeout, in milliseconds, to wait for a response from the SNMP helper.
OID|MIB_variable	Optional; the MIB node from which to begin the SNMP walk. The node can be expressed as an OID (for example, 1.3.6.1.2.1.2.2.1.1), or as a MIB variable (for example, ifIndex). If you do not specify a starting node, the script performs an SNMP walk of the entire device, starting at the 'internet' MIB node.
`-quiet`	Optional; outputs only retrieved data and essential information.
community_add_on	Optional; additional context to the community string.

Upgrade and backup scripts

These scripts are used as part of the process of upgrading and migrating from previous Network Manager versions. You can also use the ITNMExportNetworkViews.pl script to back up your network views.

ITNMDataExport.pl

This script exports Network Manager configuration data. You can then import this data into a new Network Manager system when migrating to that new system.

Running the script

Ensure the NCHOME environment variable is set on the system.

To run the script, use a command line similar to the following:

```
perl $NCHOME/precision/scripts/
upgrade/ITNMDataExport.pl -export -from 4.1.1
```

Command-line options

The following table describes the command-line options for the script.

Table 599. ITNMDataExport.pl command-line options	
Command-line option	**Description**
-export	Mandatory. Exports configuration data.
-from	Mandatory. Specifies which version of Network Manager the data is being exported from. Allowable values are version numbers in the format v.r.m (version.release.modification), separated by dots and without a leading letter V. For example: • 4.2.0 • 4.1.1 • 4.1.0 • 3.9.0
-help	Optional. Displays usage information.

ITNMDataImport.pl

This script imports Network Manager configuration data.

Running the script

Ensure the NCHOME environment variable is set on the system. The compressed configuration data exported from a previous Network Manager system must be in `NCHOME/var/precision/export`.

To run the script, use a command line similar to the following:

```
perl $NCHOME/precision/scripts/
upgrade/ITNMDataImport.pl -import -from 3.9
```

Command-line options

The following table describes the command-line options for the script.

Table 600. ITNMDataImport.pl command-line options	
Command-line option	**Description**
-import	Mandatory. Imports configuration data.
-from	Mandatory. Specifies which version of Network Manager the data is being imported from. • 3.9 • 4.1 • 4.1.1 • 4.2

Table 600. ITNMDataImport.pl command-line options (continued)	
Command-line option	**Description**
-help	Optional. Displays usage information.
-simulate	Optional. Simulates a data import. Shows what would be done, without importing data.

Importing and exporting network views using the ITNMExportNetworkViews.pl script

Use the **ITNMExportNetworkViews.pl** Perl script to back up user created network views and filter data. Export the views to a file and import the views if they become corrupted.

Description

This script backs up all user created **Network Views** in a particular domain to a file. If your **Network Views** are deleted, you can use the script to restore the **Network Views**. Do not use the script to import **Network Views** if you have existing views: you might get unpredictable results such as duplicate views. To import views, first delete existing views.

Restriction: Views that were automatically created based on the Dynamic View template are not exported. These views are automatically recreated on the target installation based on the last discovered topology.

Running the script

To run the script, use a command line similar to the following:

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/
upgrade/ITNMExportNetworkViews.pl
```

Command-line options

The following table describes the command-line options for the script.

Table 601. ITNMExportNetworkViews.pl command-line options	
Command-line option	**Description**
-export	Required. Exports network views for use on another Network Manager system. You must use either the `-export` or the `-import` option, but not both at the same time.
-import	Required. Imports network views previously exported from another Network Manager system. You must use either the `-export` or the `-import` option, but not both at the same time.
-server	Required. The type of database. Allowed values are: oracle, db2.
-dbname	Required. The name of the database, Oracle Service Name.
-host	Required. The host name of the database server.
-username	Required. User name for accessing the database.
-password	Required. Password for accessing the database.

Table 601. ITNMExportNetworkViews.pl command-line options (continued)	
Command-line option	**Description**
-port	Optional. Database port (if not using the default).
-ncimSchema	Optional. The name of the NCIM database schema (if not using the default).
-help	Optional. Displays help information for the script.
-domain	Optional. The domain to import the views into. The default is NCOMS.
-fromDomain	Optional. When exporting views, the domain to export from.
-toDomain	Optional. When importing views, the domain to import to.
-noDefaultDomain	Optional. Instructs the script not to export the network view container of the default domain that was created by the Network Manager installer.
-allocateNewEntityIds	Optional. Allocate new entity IDs for devices. If not specified, entity IDs are preserved.
-oracleService	Optional. Specifies that the Oracle database uses clustering technology and that the system must connect to the Oracle service name when Oracle clustering is configured. This option takes the following values: • 1: The installed Oracle database uses clustering technology. Connect to the Oracle service name when Oracle clustering is configured. • 0: The installed Oracle database does not use clustering technology.

ncp_ncim_diff.pl

This script identifies the differences between a customized NCIM database schema and the default NCIM schema on installation. This is useful if you are upgrading to a later version of Network Manager and you have made custom changes to the previous NCIM database schema. Once the script has identified these differences, you can manually update the NCIM schema. Using this script you can also dump the structure of an NCIM database in a specified domain to a file in XML format. You can also use the script to compare the contents of a file dump generated by this script to the structure of an NCIM database in a different domain.

Description

If your deployment requires additional network domains, you must configure process control for the domains and register the domains with the NCIM topology database. Once you have done this, you can then use the **domain_create.pl** Perl script to migrate the configuration and network polls from an existing domain to the new domain. You must use one instance of ncp_ctrl to run and manage each domain. The script does not migrate the topology from the original domain.

Running the script

To run the script, use a command line similar to one of the following commands:

Compare the structure of the NCIM database on the specified domain to the default NCIM structure on the current system.

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_ncim_diff.pl -domain NCOMS1
```

Dump the structure of the NCIM database in the specified domain to a file in XML format

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_ncim_diff.pl -domain NCOMS1 -dumpToFile NCIM_NCOMS1.xml
```

Compare the contents of a file dump generated by this script to the structure of an NCIM database in a different domain

```
$NCHOME/precision/bin/ncp_perl $NCHOME/precision/scripts/perl/scripts/
ncp_ncim_diff.pl -domain NCOMS2 -file NCIM_NCOMS1.xml
```

Command-line options

The following table describes the command-line options for the script.

Table 602. ncp_ncim_diff.pl command-line options

Command-line option	Description
`-domain` *Domain*	Mandatory; a domain with whose NCIM structure you want to perform one of the following actions: • Compare with the default NCIM structure on the current system. The script performs this action if neither the `-file` nor the `-dumpToFile` options are specified. • Compare with another NCIM structure codified in a specified XML file. The script performs this action if the `-file` option is specified. • Dump to XML file. The script performs this action if the `-dumpToFile` option is specified.
`-file` *Filename*	Optional; file in XML format describing the structure of an NCIM database on a particular domain. The script compares the NCIM structure in the domain specified with the NCIM structure described in this file. The file specified here must have been created previously using this script. **Note:** If you use this option, then you cannot use the `-dumpToFile` option.
`-dumpToFile` *Filename*	Optional; dump the structure of the NCIM database in the specified domain to the named file. **Note:** If you use this option, then you cannot use the `-file` option.
`-verbose`	Optional; provides more information on the screen.
`-help`	Optional; provides help on this command

Table 602. ncp_ncim_diff.pl command-line options (continued)	
Command-line option	**Description**
The following rows list optional arguments for connecting to the NCIM database on the specified domain. By default, the script uses the values in the `DbLogins.`*`DOMAIN`*`.cfg` configuration file for the domain to connect to the database. Any or all of the arguments below can be used to override the values in the `DbLogins.`*`DOMAIN`*`.cfg` configuration file.	
`-password` *password_for_DB_access*	Optional; almost always required, as it is usually encrypted in the DbLogins file.
`-server` db2 | oracle	Optional; type of database server. This must be one of the values shown.
`-host` *DB_server*	Optional; host name or IP address of the device running the database server.
`-port` *DB_server_port_number*	Optional; if not supplied and not read from the `DbLogins.`*`DOMAIN`*`.cfg` configuration file, then the default port number for the server type is used.
`-username` *username_for_DB_access*	Optional; username for database access.
`-schema` *schema_name*	Optional; schema name. This is usually `NCIM`.
`-dbname` *DB_name*	Optional; database name. This argument is only really meaningful for Db2, or Oracle servers.

nmExport

Use this script to export customized data for the core components from a previous version of Network Manager.

The script creates a `.pkg` file containing discovery configuration data, network views, poll policies and definitions, other configuration files, and log files containing information about the successful export. If the export is unsuccessful, log files are saved to the `/itnmExportLogs/` directory in your home directory.

Running the script

To run the script, go to the location on the source server where you extracted the export package `ExportPackage.tar`, change to the `scripts` directory at that location, and use a command line similar to the following:

```
./nmExport -n netcool_location -o Location_of_package
```

Command-line options

The following table describes the command-line options for the **`nmExport`** script.

Table 603. nmExport command-line options	
Command-line option	**Description**
`-l` *directory*	Optional; Defines where to store log files for the export operation triggered by this script. If the directory does not exist then it will be created. The default is `$HOME/itnmExportLogs/`.

Table 603. nmExport command-line options (continued)	
Command-line option	**Description**
-n *netcool_location*	Optional; location of the Network Manager installation to be migrated. If no value is provided, the environment variable $NCHOME is used; if it does not exist, then the script prompts for a value.
-o *location of package*	Optional; full path to and filename of the package containing the exported data.

nmGuiExport

Use this script to export customized GUI data from a previous version of Network Manager.

The script creates a `data.zip` file in one of the following directories: `$TIPHOME/profiles/TIPProfile/output/` or `$JazzSM_HOME/ui/upgrade/data/`. The file contains configuration data, user roles, and custom Tivoli Integrated Portal pages, views, and roles. Any errors are saved to one of the following directories: `$TIPHOME/profiles/TIPProfile/logs/tipcli.log` or `$JazzSM_HOME/ui/logs/consolecli.log`.

Running the script

To run the script, go to the location on the source server where you extracted the export package `ExportPackage.tar`, change to the `scripts` directory at that location, and use a command line similar to the following:

```
nmGuiExport -u GUI_admin_user_name -p password [ -d $TIPHOME|$JazzSM_HOME
 ]
[ –n $NMGUI_HOME ]
```

Command-line options

The following table describes the command-line options for the **nmGuiExport** script.

Table 604. nmGuiExport command-line options	
Command-line option	**Description**
-u *GUI_admin_user_name*	Either the Tivoli Integrated Portal administrative user name (tipadmin by default) for versions of Network Manager prior to V4.2, or the Jazz for Service Management administrative user name (smadmin by default). If no value is provided, then the script prompts for a value.
-p *password*	Password for the GUI administrator user. If no value is provided, then the script prompts for a value.
-d *$TIPHOME\|$JazzSM_HOME*	$TIP_HOME is the location of the Tivoli Integrated Portal installation to be migrated. For example: `/opt/IBM/tivoli/tipv2`. $JazzSM_HOME is the location of the Jazz for Service Management installation to be migrated. For example: `/opt/IBM/JazzSM`. If no value is provided, the script uses the locations set by the appropriate environment variable. If the environment variables are not set, the script prompts for a value.

Table 604. nmGuiExport command-line options (continued)	
Command-line option	**Description**
-n $NMGUI_HOME	$NMGUI_HOME is the environment variable that defines where the Network Manager GUI is installed. In Network Manager V4.2, this location is `/opt/IBM/netcool/gui/precision_gui` by default. In Network Manager V4.1 the equivalent of $NMGUI_HOME is $NCHOME, which is `/opt/IBM/netcool/core` by default. If no value is provided, the script uses the locations set by the appropriate environment variable. If the environment variable is not set, the script prompts for a value.

nmGuiImport

Use this script to import customized GUI data from a previous version of Network Manager.

Running the script

To run the script, go to the location on the target server where you extracted the `ExportPackageGUI.tar` file, change to the `scripts` directory at that location, and use a command line similar to the following:

```
nmGuiImport -u smadmin -p password [ -f path_to_zip_file ] [ -d $JazzSM_Home ]
[ –n $NMGUI_HOME ]
```

Command-line options

The following table describes the command-line options for the **nmGuiImport** script.

Table 605. nmGuiImport command-line options	
Command-line option	**Description**
-u	The Jazz for Service Management administrative user name (smadmin by default). If no value is provided, then the script prompts for a value.
-p *password*	Password for the Jazz for Service Management administrator user. If no value is provided, then the script prompts for a value.
-f *path_to_zip_file*	The path to the `data.zip` export file that contains the GUI data. If this option is not specified, the value in the `$NMGUI_HOME/integration/plugins/ITNM_42.properties` file is used.
-d *$JazzSM_Home*	$JazzSM_HOME is the location of the Jazz for Service Management installation to be migrated. For example: `/opt/IBM/JazzSM`. If no value is provided, the script uses the location set by the environment variable. If the environment variable is not set, the script prompts for a value.

Table 605. nmGuiImport command-line options (continued)	
Command-line option	**Description**
-n *$NMGUI_HOME*	The location of the Network Manager GUI components. By default, this location is `/opt/IBM/netcool/gui/precision_gui`. If no value is provided, the script prompts for a value.

nmImport

Use this script to import customized core components data from a previous version of Network Manager.

Running the script

To run the script, go to the location on the target server where you extracted the `ExportPackageCore.tar` file, change to the `scripts` directory at that location, and use a command line similar to the following:

```
./nmImport -n netcool_location
```

Note: The script asks various questions as part of execution, including the following:

- Enter all migration packages to process.
- Check current server settings.
- Specify NCIM database account password.
- In the case of copying one current system to another: Specify whether to allocate new entity identifiers.

Command-line options

The following table describes the command-line options for the **`nmImport`** script.

Table 606. nmImport command-line options	
Command-line option	**Description**
-n *netcool_location*	Optional; location of the Network Manager installation to which the data is to be migrated. If no value is provided, the environment variable $NCHOME is used; if it does not exist, then the script prompts for a value.

Chapter 30. Web Applications configuration reference

Use this information to support configuration of web applications.

Web application configuration files

There are separate configuration files for the Network Manager web applications and for Topoviz. This section explains how to change configuration settings in these files.

There are two configuration files. These files are located at `$NMGUI_HOME/profile/etc/tnm/`. These files contain all the settings used by the web applications. The configuration files are the following:

- `topoviz.properties`: Contains settings used by Topoviz.
- `status.properties`: Contains status display settings for Topoviz and the Structure Browser.
- `tnm.properties`: Contains settings used by all the other Network Manager web applications.

To change any of the settings in these files, edit the appropriate file.

Backup copies of both of these files, containing default settings, are held at the following location:`$NMGUI_HOME/profile/etc/tnm/default`.

Note: The `tnm.properties`, `status.properties`, and `topoviz.properties` files are monitored every 60 seconds for changes, so these changes are automatically detected by Topoviz.

Topoviz configuration files

The configuration files for IBM Tivoli Network Manager IP Edition Web applications and Topoviz contain all the settings that are used by each application.

The configuration files are located at `$NMGUI_HOME/profile/etc/tnm/`. The files are as follows:

- `topoviz.properties`: Contains settings used by Topoviz.
- `status.properties`: Contains status display settings for Topoviz and the Structure Browser.
- `tnm.properties`: Contains settings used by all the other Network Manager Web applications.

To change any of the settings in these files, edit the appropriate file.

Backup copies of both of these files, containing default settings, are stored at the following location: `$NMGUI_HOME/profile/etc/tnm/default/`.

The `tnm.properties`, `status.properties`, and `topoviz.properties` files are monitored every 60 seconds for changes, so that these changes are automatically detected by Topoviz.

WebTools configuration files

Each web tool has its own XML configuration file.

These configuration files are held at the following locations: `$NMGUI_HOME/profile/etc/tnm/` and `ITNMHOME/scripts/webtools/etc`.

The following table lists the configuration files.

Table 607. WebTool Configuration Files		
Type of Tool	**Name of Tool**	**Name of Associated Configuration File**
General Diagnostic and Information Retrieval Tools	Advanced Ping Tool	AdvancedPing.xml
	Advanced Subnet Ping Tool	AdvancedSubnetPing.xml
	Advanced Traceroute Tool	AdvancedTraceroute.xml
	Whois Lookup Tool	WhoisLookup.xml
	DNS Lookup Tool	DNSLookup.xml
Cisco Information Retrieval Tools	Cisco Information Tool	CiscoBGPInfo.xml
		CiscoInterfaceList.xml
		CiscoISISInfo.xml
		CiscoMBGPInfo.xml
		CiscoMPLSInfo.xml
		CiscoOSPFInfo.xml
		CiscoRoutingInfo.xml
		CiscoVRFList.xml
Cisco Diagnostic Tools	Cisco Route Information Tool	CiscoShowRoute.xml
	Cisco VRF Information Tool	CiscoVRFInfo.xml
	Cisco Ping Tool	CiscoPing.xml
	Cisco LSP Ping Tool	CiscoLSPPing.xml
	Cisco VRF Ping Tool	CiscoVRFPing.xml
	Cisco Traceroute Tool	CiscoTraceroute.xml
	Cisco LSP Traceroute Tool	CiscoLSPTraceroute.xml
	Cisco VRF Traceroute Tool	CiscoVRFTraceroute.xml
Juniper Information Retrieval Tools	Juniper Information Tool	JuniperBGPInfo.xml
		JuniperInterfaceList.xml
		JuniperISISInfo.xml
		JuniperMPLSInfo.xml
		JuniperOSPFInfo.xml
		JuniperRoutingInfo.xml
		JuniperVRFList.xml

Table 607. WebTool Configuration Files (continued)		
Type of Tool	**Name of Tool**	**Name of Associated Configuration File**
Juniper Diagnostic Tools	Juniper Route Information Tool	`JuniperShowRoute.xml`
	Juniper Ping Tool	`JuniperPing.xml`
	Juniper Traceroute Tool	`JuniperTraceroute.xml`

WebTools configuration files define all parameters used by WebTools. You should not normally need to change these parameters.

The only parameters that you might be required to configure are the Telnet login details for the Cisco and Juniper tools, as described in "Configuring Telnet Login Details".

Note: Passwords specified in the WebTools configuration files are plain-text. If you configure Telnet login details within these files, then it is recommended that you apply appropriate security measures to the WebTools directories, `$NMGUI_HOME/profile/etc/tnm/` and `ITNMHOME/scripts/webtools/etc`.

Structure Browser configuration files

The appearance and tools of the Structure Browser are controlled through configuration files.

The following configuration files control the appearance of the **Structure Browser** window. Both files are located in `$NMGUI_HOME/profile/etc/tnm/`.

- The `structurebrowser.properties` file controls settings that are only related to the **Structure Browser** window.
- The `status.properties` file controls all status indicator settings for both the **Topoviz** views and the **Structure Browser** window.
- The `ncp_structurebrowser_menu.xml` file controls what tools are available in through the Structure Browser.

Note: The `structurebrowser.properties`,`status.properties`, and `ncp_structurebrowser_menu.xml` files are monitored every 60 seconds for changes, so these changes are automatically detected by the Structure Browser.

The managed status column can be hidden or displayed, and the managed and unmanaged icons are customizable also. You can set whether the column appears in the **Device Structure Tree** and change the icons for the managed and unmanaged states in the `structurebrowser.properties` file.

URL parameters

Use URL parameters to construct a URL to launch any of the Network Manager Web applications directly from a Web browser. For example, you can construct a URL to launch the Hop View containing a predefined network map.

These parameters can be typed directly into the address bar of your browser. Alternatively, you could write a Tivoli Netcool/OMNIbus Web GUI tool to pass column values for an event to a CGI script. The script could then call the relevant Web application with these parameters.

Default windows composed of multiple Web applications, such as the Network Health View, cannot be opened using a URL. The following table lists the Network Manager Web applications that can be opened using URLs.

Table 608. GUI windows that can be opened with URLs		
Window	**URL**	**Takes parameters**
Hop View	https://host:port/ibm/console/ncp_topoviz/HopView.do	Yes
MIB Browser	https://host:port/ibm/console/ncp_mibbrowser/Launch.do	Yes
MIB Grapher	https://host:port/ibm/console/ncp_mibbrowser/pages/mib_graph/mibgraphview_servlet.jsp	Yes
Network Discovery Configuration	https://host:port/ibm/console/ncp_disco/DiscoConfig.do	No
Network Discovery Status	https://host:port/ibm/console/ncp_disco/DiscoStatus.do	No
Network Views	https://host:port/ibm/console/ncp_topoviz/NetworkView.do	Yes
Path Views	https://host:port/ibm/console/ncp_topoviz/PathViewNewPath.do	Yes
Top Performers	https://<server>:<port>/ibm/console/NetworkHealth/pages/performance/nmPerformance.jsp	Yes
Structure Browser	https://host:port/ibm/console/ncp_structureview/Launch.do?	Yes
Web Tools	https://host:port/ibm/console/ncp_webtools/pages/ncp_wt_index.jsp?	Yes

The following topics explain the URL parameters to use for the different Web applications.

Hop View URL parameters

Use this information to understand how to construct a URL to display layer 1, layer 2, and layer 3 connectivity maps in the Network Hop View.

URL parameters

The following table shows the URL parameters that you can pass to the Hop View to display layer 2 or layer 3 connectivity maps.

Note: If you specify a seed that has no matches or multiple matches, the search dialogue is displayed. Multiple matches can occur if no domain is specified and a device with the same name or IP address exists in more than one domain. Multiple matches can also occur if a domain is specified and more than one device in that domain has the same name.

Table 609. URL parameters for Hop View maps

Parameter	Description	Required?
`connectivity`	This takes an integer or string (case-sensitive) value corresponding to the entityType of the connectivity. Only entityTypes of type Topology are allowed, for example, 71 (Layer 1 Topology). This parameter corresponds to the **Connectivity** field in the Hop View toolbar. Allowed values for the `connectivity` parameter are given in "Connectivity parameters" on page 966. The default value is the default for the Hop View, or 72 (`layer 2`) if no default is configured.	No
`domain`	The name of the Network Manager domain. This corresponds to the **Domain** field in the Hop View toolbar. If you do not specify a domain, all domains are searched.	No
`endNodes`	This can take one of the following values: • `true`: Show end nodes in the map. • `false`: Do not show end nodes in the map. The default is `false`.	No
`hops`	This is the number of hops from the seed device. This corresponds to the **Hops** field in the Hop View toolbar. The default is 1.	No
`layout`	This can be any of the following: • hierarchical • symmetric • orthogonal • circular The default is `symmetric`.	No
`seed`	An identifier for the seed device. This may be the `EntityName`, `IPAddress` or `EntityId` of the required seed device. This corresponds to the **Seed** field in the Hop View toolbar. You can use multiple seed devices in the same Hop View.	Yes

Example 1: URL for layer 2 connectivity map

The following example shows the format of a Topoviz URL for a layer 2 connectivity map. Note that Topoviz URLs are case-sensitive.

```
https://host:port/ibm/console/ncp_topoviz/HopView.do?domain=MPLSTEST&type=layer2
&layout=hierarchical&seed=lon-core-cis-h.ibm.com&hops=2&endNodes=true
```

Example 2: URL for layer 3 connectivity map

The following example shows the format of a Topoviz URL for a layer 3 map. Note that Topoviz URLs are case-sensitive.

```
https://host:port/ibm/console/ncp_topoviz/HopView.do?domain=MPLSTEST&type=layer3
&layout=symmetric&seed=lon-core-cis-h.ibm.com&HOPS=2&endNodes=true
```

Example 3: URL for layer 2 connectivity map with multiple seed devices

The following example shows the format of a Topoviz URL for a layer 2 map with two devices. Note that Topoviz URLs are case-sensitive.

```
https://host:port/ibm/console/ncp_topoviz/HopView.do?domain=MPLSTEST&type=layer2
&layout=hierarchical&seed=lon-core-cis-h.ibm.com&seed=dub-core-cis-g.ibm.com
&hops=2&endNodes=true
```

```
https://host:port/ibm/console/ncp_topoviz/HopView.do?domain=MPLSTEST&type=layer3
&layout=symmetric&seed=lon-core-cis-h.ibm.com&HOPS=2&endNodes=true
```

Connectivity parameters

The following table shows the integer and string values that you can use for the `connectivity` parameter in Topoviz URLs.

Table 610.

Integer	String	Description
-1	ipsubnets	Logical collection that lists the IP address in a class A, B, or C subnet.
71	layer1	Grouping of connections which belong to a Layer 1 topology.
72	layer2	Grouping of connections which belong to a Layer 2 topology.
73	layer3	Grouping of connections which belong to a Layer 3 meshed topology.
74	convergedtopology	Based on data available in NCIM, groups together connections at the lowest layer for which data is available.
75	mplste	Grouping of connections which belong to an MPLS TE topology.
77	pseudowire	Grouping of connections which belong to a Pseudo Wire topology.
78	OSPF	Represents an OSPF topology.
81	pim	Represents PIM topologies.
83	ipmroute	Represents an IP Multicast Routing topology.
86	microwave	Represents a microwave topology.

Table 610. (continued)		
Integer	**String**	**Description**
87	logicalran	Represents a radio access network topology.
90	ltecontrolplane	Represents the devices and connectivity that make up the LTE control plane.
91	lteuserplane	Represents the devices and connectivity that make up the LTE user plane.
92	probe	Represents the source/target connectivity for an IP SLA probe.

MIB Browser URL Reference

You can launch the MIB Browser directly from a web browser. The URL required to launch an empty MIB Browser is as follows:

`https://`*`host`*`:`*`port`*`/ibm/console/ncp_mibbrowser/Launch.do`

In this URL:

- *`host`* is the IP address of the host on which the Dashboard Application Services Hub server is running.
- *`port`* is the port to access on the host on which the Dashboard Application Services Hub server is running. By default this is 16316.

This URL opens the MIB Browser with the **Domain** option menu set to the first value in the list, and no **Host** or **OID** values set in the SNMP Query toolbar.

URL Parameters

You can supply the following optional parameters when you launch the MIB Browser:

- `domain`: name of the Network Manager domain to use to obtain the MIB and SNMP data. The value of this parameter is used to set the **Domain** option menu in the Configuration Toolbar.

  If you are writing a tool to launch the MIB Browser from the **Event Viewer**, then you may wish to specify the name of the ObjectServer rather than the name of the Network Manager domain. Do this by supplying the parameter `$selected_rows.ServerName`, where `ServerName` is the field in the **Event Viewer** event that specifies the name of the ObjectServer.
- `host`: IP address of the target device to be queried for SNMP data. This value is used to populate the **Host** field in the SNMP Query Toolbar.
- `variable`: the MIB object to query. This value can be the OID of the MIB object, such as `1.3.6.1.2.1.1.3` or it can be the name of the MIB object, such as `sysUpTime`. This value is used to populate the **OID** field in the SNMP Query Toolbar.
- `resultsOnly`: takes one of the values `true` or `false`.
  - If `true`, then the MIB Browser is launched in full mode.
  - If `false`, then the MIB Browser is launched in results-only mode.

If you supply the *`domain`*, *`host`*, and *`variable`* parameters, then the MIB Browser launches, automatically performs the SNMP query specified by these parameters, and then displays the results in the SNMP Query Results Area. The type of SNMP query performed varies depending on the value of the *`variable`* parameter:

- If the *`variable`* parameter is a single MIB object in the MIB tree then the MIB Browser performs an SNMP `Get` query on startup.

- If the *variable* parameter is a table in the MIB tree then the MIB Browser performs an SNMP `Get Table` query on startup.
- In all other cases, the MIB Browser performs an SNMP `Walk` query on startup.

Examples of URLs

Some examples of URLS to launch the MIB Browser are shown below:

- `https://host:port/ibm/console/ncp_mibbrowser/Launch.do`

  The MIB Browser opens up with the **Domain** option menu set to the first value. No host or OID values are set in the SNMP Query Toolbar.
- `https://host:port/ibm/console/ncp_mibbrowser/Launch.do?domain=NCOMS`

  The MIB Browser opens up with the **Domain** option menu set to the specified domain.
- `https://host:port/ibm/console/ncp_mibbrowser/Launch.do?domain=NCOMS &host=198.162.3.4`

  The MIB Browser opens up with the **Domain** option menu set to the specified domain and the Host field set to `198.162.3.4`.
- `https://host:port/ibm/console/ncp_mibbrowser/Launch.do?domain=NCOMS &host=198.162.3.4&variable=ifTable`

  The MIB Browser opens up with the **Domain** option menu, the **Host** and **OID** fields set accordingly. In addition, an SNMP `Get Table` query will automatically be issued for the MIB object `ifTable`. The results will be displayed in the SNMP Query Results Area.
- `https://host:port/ibm/console/ncp_mibbrowser/Launch.do?domain=NCOMS &host=198.162.3.4&variable=sysUpTime&resultsOnly=true`

  The MIB Browser opens up with the **Domain** option menu, the **Host** and **OID** fields set accordingly. In addition, an SNMP `Get` query will automatically be issued for the MIB object `sysUpTime`. The MIB Browser opens in results-only mode and contains only the results showing the value of sysUpTime for the network device with IP address `198.162.3.4`.

MIB Grapher URL Reference

You can graph MIB variables for a node or interface by specifying a URL in your Web browser.

The following table shows the parameters for the MIB Grapher.

Table 611. URL parameters for the MIB Grapher

Parameter	Description	Required?
`domain`	The name of the Network Manager domain.	Yes
`host`	The hostname of the node or interface for which you want to graph MIB variables.	Yes
`init`	Required to be set to `true` for launching this window.	Yes

Example: URL to graph MIB variables for a device

https://host:port/ibm/console/ncp_mibbrowser/pages/mib_graph/mibgraphview_servlet.jsp?init=true&domain=NCOMS&host=192.168.0.2

Network Views URL parameters

You can open specific network views by using URL parameters.

URL parameters

The following table shows the URL parameters that you can pass to the network views.

Table 612. URL parameters for network views

Parameter	Description	Required?
`id`	Each saved network view has a unique ID. To find out the ID of a particular view, hover the cursor over the name of the view in the navigation tree. The ID is displayed in the status bar at the bottom of the browser window, and as a tooltip. Passing a network view ID in the URL to Network Views opens that network view. The view is shown without the navigation tree.	Yes
`networkViewDefaultTab`	Fix Pack 9 Set the property to `bookview` to have the Bookmarks tab displayed by default when the network view opens. Set the property to `netview` to have the Libraries tab displayed by default.	No
`selectNode`	This is the entity ID of an entity to which you want the display to zoom in. If you specify this parameter, it enables the Toggle Overview button on the toolbar.	Optional
`showTree`	By default, this is set to true to display the network view tree as well as network view. You can set this to false to stop the network view tree from being displayed.	Yes

URL for a saved network view

The following example shows the format of a Network Views URL containing the parameter `id`. Note that Topoviz URLs are case-sensitive.

```
https://host:port/ibm/console/ncp_topoviz/NetworkView.do?id=10690
```

Top Performers URL parameters

Use this information to construct a URL to display the historical performance data for a network view, path view, or device ID, interface ID, or both. The historical performance data displays in a tab or window that is called **Top Performers**.

URL parameters

The following table shows the parameters for the **Top Performers** URL.

*Table 613. URL parameters for the **Top Performers** URL.*			
Parameter	**Description**	**Allowed Values**	**Required?**
`viewId`	The path view ID or the network view ID, for which you want to plot the performance information. Cannot use this parameter in partnership with the parameters `entityIds` or `mainNodeEntityIds`	A positive Integer that is greater than 0.	Yes, if you want to plot information for `viewId`.
`mainNodeEntityIds`	A comma-separated list of device IDs to render the performance of the entire devices' membership. You must use this parameter in partnership with parameter `entityIds`	1 or more comma-separated positive integers. Or, to represent that there are no devices in the selection ensure that you do not enter a value.	Yes, if you want to plot information for the devices full containment.
`entityIds`	A comma-separated list of specific `entityIds` for which you want to plot the performance information. `entityIds` can represent interfaces or a device chassis. You must use this parameter in partnership with the parameter `mainNodeEntityIds`	1 or more comma-separated positive integers. Or, to represent that there are no devices in the selection ensure that you do not enter a value.	Yes, if you want to plot information for specific chassis or interfaces, or both.

Sample URL parameters

- URL for a `viewId`.

  Use this type of URL to plot a performance chart based on a defined network `viewId`.

  ```
  https://<server>:<port>/ibm/console/NetworkHealth/pages/performance/
  nmPerformance.jsp?viewId=348
  ```

- URL for specific devices.

  Use this type of URL to plot a performance chart based on a defined comma-separated list of device IDs, this URL renders the performance information of the entire devices' membership.

  ```
  https://<server>:<port>/ibm/console/NetworkHealth/pages/performance/
  nmPerformance.jsp?mainNodeEntityIds=11596,11628,11551&entityIds=
  ```

- URL for pairing interfaces.

Use this type of URL to plot a performance chart based on a defined comma-separated list of interface IDs or just the device, exclusive of its containment, this URL renders the performance information of the specific interfaces.

```
https://<server>:<port>/ibm/console/NetworkHealth/pages/performance/
nmPerformance.jsp?mainNodeEntityIds=&entityIds=12343,12785
```

Structure Browser URL reference

When you integrate other products with Network Manager, you can start the Network Manager Structure Browser directly from a web browser for that device or interface.

Use the following URL to start an empty Structure Browser. `https://host:port/ibm/console/ncp_structureview/Launch.do?`. Then, apply the following parameters to start the Structure Browser based on either the ID or the name of the device or interface for a specific domain. Use the `selectedEntityId` or `selectedEntityName` parameters to drill into a device containment and select a contained entity such as a VLAN object or an interface.

Structure Browser URL parameters

domain
The domain to which this entity belongs.

entityId
The ID of the entity to consider as the root of the structure view tree.

entityName
The name of the entity to consider as the root of the structure view tree. If you intend on using this parameter, you must also supply the `domain`.

selectEntityId
The ID of the selected entity in the structure view tree. This entity might be an interface that is selected on the overall device containment.

selectEntityName
The name of the selected entity in the structure view tree. This entity might be an interface that is selected on the overall device containment.

tableMode
The mode in which the Structure Browser opens, when it opens in table mode. Allowed values are: `devicetable`, `interfacestable`, `connectivitytable`, `nodeToNodeConnections`.

viewMode
The mode in which the Structure Browser opens. Allowed values are: `tree` or `table`.

Structure Browser URL examples

- Example URL for a Structure View of device by ID.

  `https://host:port/ibm/console/ncp_structureview/Launch.do?entityId=9`

- Example URL for a Structure View of device by name and domain.

  The `domain` value must be supplied when you start the Structure Browser by `entityName`.
  `https://host:port/ibm/console/ncp_structureview/Launch.do?entityName=ny-p1-cr28.na.test.lab&domain=AUTO`

- Example URL for a Structure View of a VLAN interface within a device by name and domain.

  This view is of the interface within the device structure. The `domain` value must be supplied when you start the Structure Browser by `entityName`.
  `https://host:port/ibm/console/ncp_structureview/Launch.do?entityName=ny-p1-cr28.na.test.lab&domain=AUTO&selectEntityName=ny-p1-cr28.na.test.lab[ Vl85 ]`

- Example URL for a Structure View of a VLAN interface by name and domain

This view is of the interface structure only. The domain value must be supplied when you start the Structure Browser by entityName.

```
https://host:port/ibm/console/ncp_structureview/Launch.do?entityName=ny-p1-
cr28.na.test.lab[ Vl85 ]&domain=AUTO&selectEntityName=ny-p1-
cr28.na.test.lab[ Vl85 ]
```

- Example URL for a Structure View of a VLAN interface within a device by ID.

```
https://host:port/ibm/console/ncp_structureview/Launch.do?
entityId=9&selectEntityId=1474
```

- Example URL for a Structure View displayed in table mode with device connectivity displayed.

```
https://host:port/ibm/console/ncp_structureview/Launch.do?
entityId=85&viewMode=table&tableMode=connectivitytable
```

Web Tools URL reference

You can also launch WebTools by specifying a URL in your web browser to call up a form-based interface. This is useful if you want to gain access to WebTools without logging into Topoviz.

To launch WebTools using a URL:

1. Open a supported web browser and enter the following URL:

```
https://host:port/ibm/console/ncp_webtools/pages/ncp_wt_index.jsp?
domain=domain name&removeHttpHeader=true
&servletDebug=false
```

   In this URL:

   - *host* is the IP address of the host on which the Dashboard Application Services Hub server is running.
   - *port* is the port to access on the host on which the Dashboard Application Services Hub server is running. By default this takes the value of 16311.
   - *domain* is the domain you want to access with the tools.

   For example,

```
https://test.itnm.com:16311/ibm/console/ncp_webtools/pages/ncp_wt_index.jsp?
domain=NCOMS&removeHttpHeader=true
&servletDebug=false
```

   The Dashboard Application Services Hub **Login** page appears in the web browser.
2. Enter your username and password.

   Note: Usernames and passwords are case-sensitive.
3. Click the **Log In** button.
   The main **WebTools** menu appears. This provides access to the following web tools:

   - General tools
   - Cisco-specific tools
   - Juniper-specific tools

Note: It is also possible to launch a web tool from a third-party application, such as a web page. To do this, launch the desired web tool in Topoviz, copy the URL that Topoviz generates in the **Address** field of your browser, and paste this URL to the third-party application.

Path Views URL parameters

You can create a new Path View or find a device in an existing Path View by specifying a URL in your Web browser.

The following table shows the parameters for Path Views.

Table 614. URL parameters for Path Views		
Parameter	**Description**	**Required?**
`domain`	The name of the Network Manager domain.	Yes, to create a path.
`entityID`	The ID of the entity to find in a path.	Yes

Example: URL to create a new path

https://host:port/ibm/console/ncp_topoviz/PathViewNewPath.do?domain=MPLSTEST&entityId=10690

Example: URL to find a device in a path

https://host:port/ibm/console/ncp_topoviz/FindPathView.do?entityId=10690

Cisco and Juniper WebTools commands

Use this information to determine which commands are executed by the Cisco and Juniper WebTools.

The following topics list the relevant commands.

Cisco information tools

Use this information to determine which commands are executed by the WebTools that retrieve information from Cisco devices.

The following table shows the commands executed by the Cisco information tools, and specifies how to launch each web tool from a network map within the Hop View or Network Views, and from the main WebTools menu.

Table 615. Cisco Information Tools Reference			
Web Tool	**Commands executed**	**Right-click menu option**	**Menu option in the main WebTools menu**

Cisco Information Tool – BGP	`show ip bgp summary`
	`show ip bgp flap-statistics`
	`show ip bgp dampened-paths`
	`show ip bgp inconsistent-as`
	`show ip bgp neighbors`
		Webtools > **Information Tools** > **View BGP Information**	**Cisco Information Tool** > **BGP information**

Cisco Information Tool – Interface	`show ip interface brief`	**Webtools** > **Information Tools** > **View Interfaces**	**Cisco Information Tool** > **Interface information**
Cisco Information Tool – ISIS	`show isis neighbors` `show isis topology`	**Webtools** > **Information Tools** > **View ISIS Information**	**Cisco Information Tool** > **ISIS information**

Table 615. Cisco Information Tools Reference (continued)

Web Tool	Commands executed	Right-click menu option	Menu option in the main WebTools menu

Cisco Information Tool – MBGP	`show ip bgp vpn all flap-statistics`
	`show ip bgp vpn all dampened-paths`
	`show ip bgp vpn all neighbors`
	`show ip bgp vpn all paths`
		Webtools > **Information Tools** > **View MBGP Information**	**Cisco Information Tool** > **MBGP information**

Cisco Information Tool – MPLS	`show ip rsvp interface`
	`show ip vrf detail`
	`show mpls l2transport vc`
	`show mpls forwarding-table`
		Webtools > **Information Tools** > **View MPLS Information**	**Cisco Information Tool** > **MPLS information**

Cisco Information Tool – MPLS TE	`show mpls traffic-eng tunnels brief` `show mpls traffic-eng autoroute`	**Webtools** > **Information Tools** > **View MPLS TE Information**	**Cisco Information Tool** > **MPLS TE Tunnel information (general)**
Cisco Information Tool – MPLS TE (filtered)	`show mpls traffic-eng tunnels source` *source* `show mpls traffic-eng tunnels destination` *destination* `show mpls traffic-eng tunnels` *tunnelInterface* `show mpls traffic-eng tunnels role [all|head|middle|remote|tail]`	**Webtools** > **Information Tools** > **View MPLS TE Information (filtered)**	Not available
Cisco Information Tool – MPLS TE Link Management	`show mpls traffic-eng link-management summary` `show mpls traffic-eng link-management interfaces [`*interface*`]`	**Webtools** > **Information Tools** > **View MPLS TE Link Management Information**	Not available

Table 615. Cisco Information Tools Reference (continued)			
Web Tool	**Commands executed**	**Right-click menu option**	**Menu option in the main WebTools menu**

Cisco Information Tool – OSPF	`show ip ospf`
	`show ip ospf interface`
	`show ip ospf neighbor`
	`show ip ospf border-routers`
	`show ip ospf statistics`
		Webtools > **Information Tools** > **View OSPF Information**	**Cisco Information Tool** > **OSPF Information**

Cisco Information Tool – Routing Summary	`show ip protocols`
	`show ip route summary`
	`show ip route static`
	`show ip route eigrp`
	`show ip route ospf`
	`show ip route isis`
		Webtools > **Information Tools** > **View Routing Summary Information**	**Cisco Information Tool** > **Routing summary**

Cisco Information Tool – VRF List	`show ip vrf list` `show ip vrf interfaces`	**Webtools** > **Information Tools** > **View VRF Information**	**Cisco Information Tool** > **VRF list**

Cisco diagnostic tools

Use this information to determine which commands are executed by the WebTools that perform diagnosis on Cisco devices.

The following table shows the commands executed by the Cisco diagnostic tools, and specifies how to launch each web tool from a network map within the Hop View or Network Views, and from the main WebTools menu.

Table 616. Cisco and Juniper WebTools Reference			
Web Tool	**Commands Executed**	**Menu Option**	**Menu Option in the main WebTools menu**
Cisco Route Information Tool	`show ip route` *`target`*	**Cisco Tools... Diagnostic Tools... View a Route...**	**Cisco Routing Information**
Cisco VRF Information Tool	`show ip route vrf` *`vrf_name target`*	Not available	**Cisco VRF Information**
Cisco Ping Tool	`ping` *`target`*	**Cisco Tools...** **Diagnostic Tools...** **Ping from this device...**	**Cisco Ping**

Table 616. Cisco and Juniper WebTools Reference (continued)			
Web Tool	**Commands Executed**	**Menu Option**	**Menu Option in the main WebTools menu**
Cisco LSP Ping Tool	`ping mpls ipv4 target verbose`	**Cisco Tools...** **Diagnostic Tools...** **LSP Ping from this device...**	**Cisco LSP Ping**
Cisco VRF Ping Tool	`ping vrf vrf_name ip target`	**Cisco Tools...** **Diagnostic Tools...** **VRF Ping from this device...**	**Cisco VRF Ping**
Cisco Traceroute Tool	`traceroute target`	**Cisco Tools...** **Diagnostic Tools...** **Traceroute from this device...**	**Cisco Traceroute**
Cisco LSP Traceroute Tool	`traceroute mpls ipv4 target verbose`	**Cisco Tools...** **Diagnostic Tools...** **LSP Traceroute from this device...**	**Cisco LSP Traceroute**
Cisco VRF Traceroute Tool	`traceroute vrf vrf_name ip target`	**Cisco Tools...** **Diagnostic Tools...** **VRF Traceroute from this device...**	**Cisco VRF Traceroute**

Juniper information tools

Use this information to determine which commands are executed by the WebTools that retrieve information from Juniper devices.

The following table shows the commands executed by the Juniper information tools, and specifies how to launch each web tool from a network map within the Hop View or Network Views, and from the main WebTools menu.

Table 617. Cisco and Juniper WebTools Reference			
Web Tool	**Commands Executed**	**Menu Option**	**Menu Option in the main WebTools menu**

Juniper Information Tool – BGP	`show ip bgp summary`
	`show ip bgp flap-statistics`
	`show ip bgp dampened-paths`
	`show ip bgp inconsistent-as`
	`show ip bgp neighbors`
		Juniper Tools...
		Information Tools...
		View BGP Information...
			Juniper Information Tool

Table 617. Cisco and Juniper WebTools Reference (continued)

Web Tool	Commands Executed	Menu Option	Menu Option in the main WebTools menu
Juniper Information Tool – Interfaces	`show ip interface brief`	**Juniper Tools...** **Information Tools...** **View Interfaces...**	**Juniper Information Tool**
Juniper Information Tool – ISIS	`show isis neighbors` `show isis topology`	**Juniper Tools...** **Information Tools...** **View ISIS Information...**	**Juniper Information Tool**

Juniper Information Tool – MPLS	`show ip rsvp interface`
	`show ip vrf detail`
	`show mpls l2transport vc`
	`show mpls forwarding-table`
		Juniper Tools...
		Information Tools...
		View MPLS Information...
			Juniper Information Tool

Juniper Information Tool – OSPF	`show ip ospf`
	`show ip ospf interface`
	`show ip ospf neighbor`
	`show ip ospf border-routers`
	`show ip ospf statistics`
		Juniper Tools...
		Information Tools...
		View OSPF Information...
			Juniper Information Tool

Juniper Information Tool – Routing Summary	`show ip protocols`
	`show ip route summary`
	`show ip route static`
	`show ip route eigrp`
	`show ip route ospf`
	`show ip route isis`
		Juniper Tools...
		Information Tools...
		View Routing Summary Information...
			Juniper Information Tool

Juniper Information Tool – VRF List	`show ip vrf list` `show ip vrf interfaces`	**Juniper Tools...** **Information Tools...** **View VRF Information...**	**Juniper Information Tool**

Juniper diagnostic tools

Use this information to determine which commands are executed by the WebTools that perform diagnosis on Cisco devices.

The following table shows the commands executed by the Juniper diagnostic tools, and specifies how to launch each web tool from a network map within the Hop View or Network Views, and from the main WebTools menu.

Table 618. Juniper Diagnostic Tools

Web Tool	Commands Executed	Menu Option	Menu Option in the main WebTools menu
Juniper Route Information Tool	`show route` *target*	**Juniper Tools...** **Diagnostic Tools...** **View a Route...**	**Juniper Routing Information**
Juniper Ping Tool	`ping` *target*	**Juniper Tools...** **Diagnostic Tools...** **Ping from this device...**	**Juniper Ping**
Juniper Traceroute Tool	`traceroute` *target*	**Juniper Tools...** **Diagnostic Tools...** **Traceroute from this device...**	**Juniper Traceroute**

Chapter 31. Report reference

Network Manager reports are grouped by their function. Use this reference to understand the typical uses, prerequisites, and other properties of each report.

Network Manager data model

Network Manager provides Cognos® data model namespaces, which contain query subjects to use to build up reports.

Namespaces

The Network Manager data model provides the following namespaces for designing reports.

Event
The Event namespace contains query subjects to create Current Status reports.

Monitoring Data
The Monitoring Data namespace contains query subjects to create Performance reports. The polled data timestamp has a time dimension relationship to allow time dimension reports. The data for the Monitoring Data namespace comes from the NCPOLLDATA database.

Network
The Network namespace contains query subjects to create Asset and Troubleshooting reports. The data for the Network namespace comes from the NCIM database.

Network Views
The Network Views namespace contains query subjects to create reports about network views and policies updating views. The data for the Network Views namespace comes from the NCPGUI and NCMONITOR databases.

Path Views
The Path Views namespace contains query subjects to create Path Views reports.

Shared
The Shared namespace contains query subjects that can be shared to prevent query subject duplicates.

Shared Dimensions
The Shared Dimensions namespace contains query subjects to create reports with Time Dimension.

Performance Value
The Performance Value namespace contains the performance value to build reports based on Time.

Monitoring Dimension
The Monitoring Dimension namespace contains the monitoring information to navigate through its attributes.

Asset reports

Asset reports provide views on the discovered attributes of the network devices for inventory information.

To access the Asset reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Asset Reports**.

Card Detail by Device Type report

Displays the results of device discovery operations performed by the entity MIB agent. The report is organized by device type and is based on the data extracted from the entPhysicalVendorType entity MIB table.

Report properties

The following table describes the Card Detail by Device Type report.

Table 619. Properties of the Card Detail by Device Type report	
Property	**Description**
Typical uses	Use this report identify the card details within a device.
Prerequisites	This report uses information from the Entity MIB, and requires the Entity discovery agent to be enabled.

Discovery report

Displays the results of the network discovery organized by device vendor. The report displays a list of vendor names. It lists device classes for each vendor.

Report properties

The following table describes the Discovery report.

Table 620. Properties of the Discovery report	
Property	**Description**
Typical uses	Use this report to look up interface details on a device, such as spare ports, or MAC and IP addresses.
Prerequisites	None

Interface Availability report

Displays a table of interface types and statuses broken down by vendor, class name and device. The report also displays the count of each interface type on the network and the status of those interfaces (up or down). One report is displayed for each selected domain.

Report properties

The following table describes the Interface Availability report.

Table 621. Properties of the Interface Availability report	
Property	**Description**
Typical uses	Use this report to determine the counts of each interface type, both functioning and non-functioning.
Prerequisites	None

IP Addressing Summary report

Displays information on the IP addresses used in the network grouped according to CIDR notations.

Report properties

The following table describes the IP Addressing Summary report.

Table 622. Properties of the IP Addressing Summary report	
Property	**Description**
Typical uses	Use this report to identify IPv4 subnets with spare IP addressing capacity or those subnets over a specific threshold of allocated IP addresses.
Prerequisites	None

Operating System by Device report

Displays information on the operating systems running on the various devices in your network. This report only shows information for Cisco and Juniper devices.

Report properties

The following table describes the Operating System by Device report.

Table 623. Properties of the Operating System by Device report	
Property	**Description**
Typical uses	Use this report to show operating system details by device and vendor. For example, you can locate devices with a certain operating system that has a newly released security update.
Prerequisites	The OSInfo agent must be run during discovery in order for the information required for this report to be available.

Context reports

Context reports show information related to the selected device.

To access Context reports, right-click a device in any topology view and select **Reports** > ***Report name***.

Bandwidth In Utilization report

Displays the SNMP in bandwidth utilization of a device.

Report properties

This is the Bandwidth Utilization report using snmpInBandwidth poll policy. The following table describes the Bandwidth In Utilization report.

Table 624. Properties of the Bandwidth In Utilization report	
Property	**Description**
Typical uses	Use this report to see the bandwidth in use of selected.

Table 624. Properties of the Bandwidth In Utilization report (continued)	
Property	**Description**
Prerequisites	To use this report, you must enable the snmpInBandwidth poll policy with the Store Poll Data option enabled.

IfInDiscards report

Displays the ifInDiscards of the device.

Report properties

This is the Cognos Generic Trend Analysis report using ifInDiscards poll policy. The following table describes the IfInDiscards report.

Table 625. Properties of the IfInDiscards report	
Property	**Description**
Typical uses	Use this report to see the trend of the discarded packets of an interface on a device.
Prerequisites	To use this report, you must enable the ifInDiscards poll policy with the Store Poll Data option enabled.

Memory usage report

Displays the memory usage of a device.

Report properties

This is the Cognos Generic Trend Analysis report using memoryPoll poll policy. The following table describes the Memory usage report.

Table 626. Properties of the Memory usage report	
Property	**Description**
Typical uses	Use this report to see the trend of the memory usage of a device.
Prerequisites	To use this report, you must enable the memoryPoll poll policy with the Store Poll Data option enabled.

CPU Usage report

Displays the CPU usage of the device.

Report properties

This is the Cognos Generic Trend Analysis report using cpuBusyPoll poll policy. The following table describes the CPU Usage report.

Table 627. Properties of the CPU Usage report	
Property	**Description**
Typical uses	Use this report to see a history of the CPU usage of a device.

Table 627. Properties of the CPU Usage report (continued)	
Property	**Description**
Prerequisites	To use this report, you must enable the cpuBusyPoll poll policy with the Store Poll Data option enabled.

Monitoring reports

Monitoring reports provide a list of devices being polled under each monitoring policy to help you verify that you are polling the correct devices for the correct information.

To access the Monitoring reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Monitoring Reports**.

Monitoring Device Details report

Displays detailed information about the monitoring policies enabled for a device. To run this report you must have poll policies with the store option enabled.

Report properties

The following table describes the Monitoring Device Details report.

Table 628. Properties of the Monitoring Device Details report	
Property	**Description**
Typical uses	You would run this report to verify how a particular device is being monitored by Network Manager by listing all the policies whose scope matches this device.
Prerequisites	To run this report you must have poll policies with the store option enabled.

Monitoring Policy Details report

Displays detailed information about a selected monitoring policy. To run this report you must have poll policies with the store option enabled.

Report properties

The following table describes the Monitoring Policy Details report.

Table 629. Properties of the Monitoring Policy Details report	
Property	**Description**
Typical uses	Run this report to verify the list of devices being monitored by this policy.
Prerequisites	To run this report you must have poll policies with the store option enabled.

Monitoring Summary report

Also known as the Monitoring Policies report, if it is launched as a right-click report. Displays all the enabled policies and for each policy all the devices and interfaces that match the scope. To run this report you must have poll policies with the store option enabled.

Report properties

The following table describes the Monitoring Summary report.

Table 630. Properties of the Monitoring Summary report	
Property	**Description**
Typical uses	You can run this report to archive the monitoring configuration for your network, or use it for reference purposes offline.
Prerequisites	To run this report you must have poll policies with the store option enabled.
Data model	Cognos

Network Technology reports

Network Technology reports provide insight into the states of BGP, OSPF, and VLAN networks based on information gathered during discovery.

Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Click **Network Technology Reports**.

BGP Details report

Displays detailed information about BGP Sessions and Autonomous Systems.

Report properties

The following table describes the BGP Details report.

Table 631. Properties of the BGP Details report	
Property	**Description**
Typical uses	Run this report to see the members of each BGP session and its state for each Autonomous System. View the details of any route reflectors and their clients and the state of each member of the Autonomous Systems.
Prerequisites	None

BGP Summary report

Displays charts and tables with BGP Session and Autonomous System Summary information.

Report properties

The following table describes the BGP Summary report.

Table 632. Properties of the BGP Summary report	
Property	**Description**
Typical uses	Run this report to see a quick count of the elements in the BGP environment including the Autonomous Systems, inter-AS, and sessions per state.
Prerequisites	None

LTE Interfaces report

Displays a list of all LTE interfaces.

Report properties

The following table describes the LTE Interfaces report.

Table 633. LTE Interfaces report	
Property	**Description**
Typical uses	Run this report to view a list of all LTE interfaces arranged according to interface type.
Prerequisites	None

MPLS VPN Details report

Displays detailed information about discovered MPLS VPNs including VRFs, Route Distinguishers, Route Targets, and VPWS.

Report properties

The following table describes the MPLS VPN Details report.

Table 634. Properties of the MPLS VPN Details report	
Property	**Description**
Typical uses	Run this report to see details of the MPLS VPNs discovered in this domain. See details of the VRF Route Targets including import/export mismatches, membership details of VPN and VPWS, as well as PE/CE connections and PE/P connections.
Prerequisites	None

MPLS VPN Summary report

Displays charts and tables with MPLS VPN summary information.

Report properties

The following table describes the MPLS VPN Summary report.

Table 635. Properties of the MPLS VPN Summary report	
Property	**Description**
Typical uses	Run this report to see a quick list and count of the VPNs, VPWS devices, and associated PE/CE devices.
Prerequisites	None

VLAN Details report

Displays detailed information about VLANs and trunk ports.

Report properties

The following table describes the VLAN Details report.

Table 636. Properties of the VLAN Details report	
Property	**Description**
Typical uses	Run this report to see a list of VLAN IDs on each interface of a VLAN supported device, or a list of interfaces in each VLAN ID.
Prerequisites	None

Network Views reports

These reports show details about network views.

To access the Network Views reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Network Views Reports**.

Monitored Network Views report

Displays the poll definitions, policies, and entities that are being monitored for each network view.

Report properties

The following table describes the Monitored Network Views report.

You can drill down from this report to see the devices and interfaces monitored by an individual poll definition in the Monitored Network Views Drilldown report.

Table 637. Properties of the Monitored Network Views report	
Property	**Description**
Typical uses	Run this report to see the poll definitions, policies, and entities that are being monitored.
Prerequisites	None

Path Views reports

Path Views reports allow you to view device and routing information for IP and MPLS TE paths.

To access the Path Views reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Click **Path Views Reports**.

IP Path Summary report

Displays all the IP Paths configured.

Report properties

The following table describes the IP Path Summary report.

Table 638. Properties of the IP Path Summary report	
Property	**Description**
Typical uses	Use this report to view a list of all user created paths showing ingress and egress information of the path, status, and path changes. From this report drill down into any of the paths to check ingress and egress interface details of each hop.
Prerequisites	None

IP Routing Info report

Displays the device and routing information for a specific device or multiple devices on the path.

Report properties

The following table describes the IP Routing Info report.

Table 639. Properties of the IP Routing Info report	
Property	**Description**
Typical uses	Generate this report from a member device of a user-created path on a topology map to see details of the ingress and egress interfaces of the device for this path.
Prerequisites	None

MPLS TE Path Summary report

Displays all the MPLS TE Tunnels that were discovered in the network.

Report properties

The following table describes the MPLS TE Path Summary report.

Table 640. Properties of the MPLS TE Path Summary report	
Property	**Description**
Typical uses	Use this report to view a list of all MPLS-TE tunnels showing ingress and egress information of the tunnel, status, and path changes. From this report drill down into any of the tunnels to check ingress and egress interface details of each hop.
Prerequisites	None

MPLS TE Routing Info report

Displays the device and routing information for a specific device or multiple devices on the tunnel.

Report properties

The following table describes the MPLS TE Routing Info report.

Table 641. Properties of the MPLS TE Routing Info report	
Property	**Description**
Typical uses	Generate this report from a member device of a MPLS-TE tunnel on any topology map to see details of the ingress and egress interfaces of the device for this tunnel.
Prerequisites	None

Performance reports

Performance reports allow you to view the last hour of performance data that has been collected by the monitoring system for diagnostic purposes. In addition, the Device Summarization, Interface Summarization, and Interface Availability Summarization reports allow you to view any historical performance data that has been collected by the monitoring system. View trend and topN charts for data to gain insight on short term behaviors.

To access the Performance reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Performance Reports**.

Note: The amount of historical data that the system can store and, consequently, the amount of historical data that the Performance reports can display, is restricted by default to preserve report performance. You can increase the storage limit for historical performance data; however, this can lead to a degradation in the performance of the Performance reports.

Bandwidth Top N report

Displays the bandwidth of the top N devices.

Report properties

The following table describes the Bandwidth Top N report.

Table 642. Properties of the Bandwidth Top N report	
Property	**Description**
Typical uses	Use this report to identify interfaces with the heaviest bandwidth use, and drill down to see the usage over time.
Prerequisites	None

Bandwidth Utilization report

Displays the bandwidth utilization of a device.

Report properties

The following table describes the Bandwidth Utilization report.

Table 643. Properties of the Bandwidth Utilization report	
Property	**Description**
Typical uses	Use this report to see the bandwidth use of selected devices, and drill down to see the usage over time per interface.
Prerequisites	None

Composite Trending report

Displays a composite chart that contains data for two poll definitions.

Report properties

The following table describes the Composite Trending report.

Table 644. Properties of the Composite Trending report	
Property	**Description**
Typical uses	Use this report to show a list of selected devices and drill down to see the trend of up to six data items.
Prerequisites	None

Device Availability Summarization

Displays a summary of device availability for the last seven days. This report uses the historical poll data tables in the NCPOLLDATA database.

Report properties

The following table describes the Device Availability Summarization report.

Table 645. Properties of the Device Availability Summarization report	
Property	**Description**
Typical uses	Use this report to see device availability data collected and summarized over the last seven days.
Prerequisites	To use this report, you must be using Apache Storm to aggregate raw poll data into historical poll data.

Device Summarization report

Displays summarization data for devices. This report uses the historical poll data tables in the NCPOLLDATA database.

Report properties

The following table describes the Device Summarization report.

Table 646. Properties of the Device Summarization report	
Property	**Description**
Typical uses	Use this report to see device level data collected and summarized over a longer period of time.
Prerequisites	To use this report, you must be using Apache Storm to aggregate raw poll data into historical poll data.
.Data model	Cognos

Historical SNMP Top or Bottom N report

Displays the top or bottom N devices with drilldown to a chart according to device or interface.

Report properties

The following table describes the Historical SNMP Top or Bottom N report.

Table 647. Properties of the Historical SNMP Top or Bottom N report	
Property	**Description**
Typical uses	Use this report to identify the best or worst performers, by average value, for any collected SNMP metric, and drill down to see the trend over time.
Prerequisites	None

Historical SNMP Trend Analysis report

Displays the device summary with drilldown to a chart according to device or interface.

Report properties

The following table describes the Historical SNMP Trend Analysis Report.

Table 648. Properties of the Historical SNMP Trend Analysis Report	
Property	**Description**
Typical uses	Use this report to see the average values collected for of a list of selected devices and drilldown to see the trend over time for that data item.
Prerequisites	None

Historical SNMP Trend Quick View report

Displays the device list with drilldown to a chart according to device or interface.

Report properties

The following table describes the Historical SNMP Trend Quick View report.

Table 649. Properties of the Historical SNMP Trend Quick View report	
Property	**Description**
Typical uses	Use this report to quickly list a set of selected devices to be used as an index to drill down to see a trend graph over time.
Prerequisites	None

Interface Availability Summarization report

Displays a summary of interface availability for the last seven days. This report uses the historical poll data tables in the NCPOLLDATA database.

Report properties

The following table describes the Interface Availability Summarization report.

Table 650. Properties of the Interface Availability Summarization report	
Property	**Description**
Typical uses	Use this report to see interface availability data collected and summarized over the last seven days.
Prerequisites	To use this report, you must be using Apache Storm to aggregate raw poll data into historical poll data.

Interface Summarization report

Displays summarization data for interfaces. This report uses the historical poll data tables in the NCPOLLDATA database.

Report properties

The following table describes the Interfaces Summarization report.

Table 651. Properties of the Interfaces Summarization report	
Property	**Description**
Typical uses	Use this report to see interface level data collected and summarized over a longer period of time.
Prerequisites	To use this report, you must be using Apache Storm to aggregate raw poll data into historical poll data.
Data model	Cognos

System Availability Summary report

Displays availability summary for devices with drilldown to a chart according to device. This report is based on the sysUptime data.

Report properties

The following table describes the System Availability Summary report.

Table 652. Properties of the System Availability Summary report	
Property	**Description**
Typical uses	Use this report to see availability statistics as defined by collected sysUptime data.
Prerequisites	None

Troubleshooting reports

Troubleshooting reports help you identify problems while optimizing the discovery of the network as well as help identify possible problems discovered in the network such as duplex mismatches.

To access the Troubleshooting reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Troubleshooting Reports**.

Connected Interface Duplex Mismatch report

Displays a list of connections where one end of the connection is half-duplex and the other end is full-duplex.

Report properties

The following table describes the Connected Interface Duplex Mismatch report.

Table 653. Properties of the Connected Interface Duplex Mismatch report	
Property	**Description**
Typical uses	Diagnosing performance or availability issues. **Tip:** The duplex value for the interfaces are learned at discovery time from the dot3StatsDuplexStatus value in the EtherLike-MIB.mib. This MIB defines the values for dot3StatsDuplexStatus as: unknown(1); halfDuplex(2); and fullDuplex(3). A value of unknown means that Network Manager cannot determine the duplex status based on the available MIB information.
Prerequisites	A completed and successful discovery with the Interface Agent enabled.

Devices Pending Delete on Next Discovery report

Displays information on devices to be deleted from the topology if they are not found during the next discovery cycle.

Report properties

The following table describes the Devices Pending Delete on Next Discovery report.

Table 654. Properties of the Devices Pending Delete on Next Discovery report	
Property	**Description**
Typical uses	If device has been removed from the network, it will remain in the topology for x more discoveries, where x is the value of the LingerTime variable for the device in the topology database. This report can show devices that you do not expect to be deleted from the topology, and you can investigate why they were not discovered.
Prerequisites	None

Devices with no SNMP Access report

Displays those devices to which the discovery could not get SNMP access.

Report properties

The following table describes the Devices with no SNMP Access report.

Table 655. Properties of the Devices with no SNMP Access report	
Property	**Description**
Typical uses	Troubleshooting devices for which no connectivity information was discovered. There might be a number of reasons why the discovery agents could not get SNMP access to these devices, for example, incorrect SNMP community strings.
Prerequisites	None

Devices with Unclassified SNMP Object IDs report

Displays those devices with SNMP Object IDs (OIDs) that have not been assigned to specific classes.

Report properties

The following table describes the Devices with Unclassified SNMP Object IDs report.

Table 656. Properties of the Devices with Unclassified SNMP Object IDs report	
Property	**Description**
Typical uses	This report shows devices that could not be classified properly by analyzing the ncim.mappings table to check whether the sysObjectId is recognized. You can then investigate whether the Active Object Classes (AOCs) need to be modified to be able to classify devices with these OIDs. For example, the correct agents might not have been run.
Prerequisites	None

Devices with Unknown SNMP Object IDs report

Displays those devices with unknown SNMP Object IDs (OIDs).

Report properties

The following table describes the Devices with Unknown SNMP Object IDs report.

Table 657. Properties of the Devices with Unknown SNMP Object IDs report

Property	Description
Typical uses	Use this report to identify devices with sysObjectId that were not recognized. For example, Network Manager might recognize the sysObjectID as belonging to a specific vendor, but not the specific model. Such devices are collected in the class NetworkDevice. Update both the AOC files and the ncim.mappings table in order to correctly classify the device.
Prerequisites	None

Utility reports

Utility reports display all discovered devices and their interfaces in different views.

To access the Utility reports, complete the following steps. Click the **Reporting** icon and select **Common Reporting**. Within the widget, select **Network Manager**. A list of folders display. These folders contain all Cognos reports for your access. Then click **Utility Reports**.

Discovered Nodes and Interfaces Flat File List report

Displays all discovered devices and interfaces.

Report properties

The following table describes the Discovered Nodes and Interfaces Flat File List report.

Table 658. Properties of the Discovered Nodes and Interfaces Flat File List report

Property	Description
Typical uses	Use this report to archive discovered devices and interfaces, or to export to a third party tool such as a spreadsheet or database.
Prerequisites	None

Notices

This information applies to the PDF documentation set for IBM Tivoli Network Manager IP Edition.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

958/NH04
IBM Centre, St Leonards
601 Pacific Hwy
St Leonards, NSW, 2069
Australia

IBM Corporation
896471/H128B
76 Upper Ground
London
SE1 9PZ
United Kingdom

IBM Corporation
JBF1/SOM1 294
Route 100
Somers, NY, 10589-0100
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The terms in Table 659 on page 997 are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Table 659. IBM trademarks

AIX®	Informix®	PR/SM
BNT	iSeries	System p
ClearQuest	Jazz	System z
Cognos	Lotus	Tivoli®
Db2	Netcool®	WebSphere
Db2 Universal Database	NetView®	z/OS®
developerWorks®	OMEGAMON®	z/VM®
DS8000	Passport Advantage	zSeries
Enterprise Storage Server®	PowerPC	
IBM	PowerVM®	

Adobe, Acrobat, Portable Document Format (PDF), PostScript, and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering may collect IP addresses, user names and passwords for the purpose of performing network discovery. Failure to enable the collection of this information would likely eliminate important functionality provided by this Software Offering. You as customer should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details, and the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/privacy.

Part Number:

Printed in the Republic of Ireland

(1P) P/N:

2021-4213-01