IBM Netcool/OMNIbus 8.1

Best Practices

Licensed Materials – Property of IBM

SECOND EDITION

Note: Before using this information and the product it supports, read the information in “Notices” located at the end of this document.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

US Government Users Restricted Rights–Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this publication ... vii
Intended audience ... vii
What this publication contains ... vii
Publications ... viii
Your IBM Netcool/OMNIbus Best Practices library ... viii
Prerequisite publications ... ix
Conventions used in this publication ... ix
Typeface conventions ... x
Operating system-dependent variables and paths ... x

Preface: The Event Service Framework (ESF) ... xi

Chapter 1 Introduction ... 1

Chapter 2 Planning ... 2
Standard Multitier Architecture Configuration ... 2
A starting point ... 3
Background: Interpreting the ObjectServer profiling log ... 3
Background: Interpreting the ObjectServer trigger statistics log ... 4
Multitiered deployment: the purpose of each tier ... 5
Use the standard multitier architecture configuration ... 14
Use the initial configuration wizard ... 14
Requirements gathering ... 14
Solution sizing and design based on requirements ... 15
Estimating the number of managed entities ... 15
Estimated number of events per day and event rate ... 16
Estimated number of Probes ... 16
Assessing Probe load on the ObjectServer ... 16
Consideration for event bursts ... 17
Estimated number of standing events in the ObjectServer ... 17
Estimated number of users ... 18
Provisioning guidelines summary ... 19
Where to implement custom functionality ... 22
ObjectServer trigger or IBM Netcool/Impact policy? ... 23
Consider using scope-based event grouping ... 24
Architectural considerations ... 25
Geographical layout ... 25
Security and network restrictions ... 25
Resiliency ... 26
ObjectServer loading due to custom functionality ... 27
Hardware considerations ... 27
Hardware resiliency ... 27
CPU, memory, and disk space resourcing ... 28
Example hardware specifications ... 30
Setting up file descriptors ... 31
Network resourcing ... 31
Historical event archive sizing guidance ... 32
Housekeeping historical event archive data ... 33
Other installation prerequisites ... 34
Solution delivery ... 35
Custom configuration ... 35
Documentation ... 37
Use of debug mode ... 38
Stress testing prior to deployment into production ... 38
Checking the table_store memstore hard and soft limits ... 38
Removal of temporary files ... 40

Chapter 3 Installing and upgrading ... 41
IBM Prerequisite Scanner ... 41
Netcool/OMNIbus WebGUI installation ... 41
Updating the ObjectServer schema when upgrading ... 41

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 4 ObjectServers 43
ObjectServer properties 43
Connections 43
Granularity 43
Iduc.ListeningPort 43
PA.Name, PA.Username, PA.Password 43
Profile 44
SecureMode 44
Default triggers 44
Do not modify default triggers 44
generic_clear trigger 45
expire trigger 47
delete_clears trigger 47
clean_details_table trigger 47
clean_journal_table trigger 47
Custom trigger groups 48
Custom triggers 48
Temporal triggers 48
Database triggers 50
Signals and signal triggers 52
Using the WHEN clause 54
Use of comments 55
Standardisation of layout and formatting 55
Avoid table scanning where possible 59
Avoid nesting for-each-row loops 59
Ordering of SQL where clauses 61
Use of the Evaluate tab 62
Default escalations 64
SuppressEscl 64
flash_not_ack 65
escalate_off 65
mail_on_critical 66
Procedures 66
SQL procedures 66
External procedures 68
Default procedures 70
automation_enable, automation_disable procedures 70
jinsert procedure 70
Indexes 71
When to use indexing 71
How to determine what effect indexing has made 71
Tools, menus, and prompts 73
Adding custom fields 74
Include custom fields in solution delivery SQL file 74
Include custom field conversions in solution delivery SQL file 75
Add custom fields to Gateway mappings 76
Create generic fields for future use 78
User and group creation 79
Process for user and group creation 80
Using VMMSYNC on WebGUI 81
IDUC port for firewall configuration 82
ObjectServer HTTP and OSLC ports 82
ObjectServer housekeeping 82
ExpireTime 83
Monitoring row numbers 86
De-escalation of events 90
Backing up the ObjectServer 93
ObjectServer data files 93
About the automated backup function 93
About the nco_osreport utility 94
Recovering an ObjectServer 95
Resynchronising a newly restored ObjectServer 95
Event flood control 96
Accelerated Event Notification (AEN) 97

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Maximum number of events in the Event List 97
Setting up support for multitenancy 98

Chapter 5 Probes 99

Probe configuration file locations 99
Back up Probe configuration files before modification 99
Create custom Probe configuration file subdirectories 99
Generic properties 100
Probe peering recommended where appropriate 100
Circular store and forward mode for Probe failover/failback 100
Enabling self-monitoring of Probes 101
Generic properties 101
dumpprops property 105
Running more than one instance of a Probe on the same host 105
Probe rules files 106
Use of comments 107
Location of customisations within the rules file 108
Use of indentation and white space 108
Code efficiency 108
Use of include files 109
Use of details 109
Name-value pair (NVP) functions and ExtendedAttr 111
Netcool Knowledge Library (NcKL) 113
MIB to rules file conversion 114
Detecting event floods and anomalous event rates 114
Probe rules file development tips and tricks 114
Debugging Probe rules files 114
RawCapture output for unparsed events 115
regmatch() before extract() 115
Obtaining sub-second timing information in the rules file 116
Use regreplace() to clean up tokens/strings 117
Using the update() function 118

Chapter 6 Gateways 119

Generic properties 119
MaxLogFileSize 119
MessageLevel 119
Gate.CacheHashTblSize 119
dumpprops property 120
File locations 120
Original files and backups 121
Gateway configuration files 121
Gateway mapping files 121
Gateway table replication files 122
Cold standby Gateways (UNIX only) 123
ObjectServer Gateway resynchronisation types 124
NORMAL resynchronisation type 124
UPDATE resynchronisation type 124
MINIMAL resynchronisation type 125
TWOWAYUPDATE resynchronisation type 126
ObjectServer Gateway resynchronisation lock types 127
FULL resynchronisation lock type 127
PARTIAL resynchronisation lock type 127
NONE resynchronisation lock type 128

Chapter 7 Anatomy of the standard multitier architecture configuration 129

Failback at Collection and Aggregation 129
Failback at the Aggregation layer 130
Failback at the Collection layer 131
File locations 132
The Collection layer 132
Collection layer ObjectServer triggers 132
The Collection Gateway properties file 134
The Collection Gateway table replication file 138
The Collection Gateway mapping file 140

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Aggregation layer ... 141
Aggregation layer ObjectServer triggers ... 141
The Aggregation failover Gateway properties file ... 142
The Aggregation failover Gateway table replication file ... 146
The Aggregation failover Gateway mapping file ... 146
The Display layer ... 147
Display layer ObjectServer triggers ... 147
The Display Gateway properties file ... 148
The Display Gateway table replication file ... 152
The Display Gateway mapping file ... 153
The TimeToDisplay field ... 153

Chapter 8 Proxy servers ... 155

Chapter 9 Firewall Bridge server ... 156

Chapter 10 The Netcool Process Agent and machine start-up ... 157

Netcool Process Agent configuration file ... 157
Machine start-up ... 157
Process Agent security considerations ... 158
Running the Netcool Process Agent as a non-privileged user (UNIX) ... 158
Running the Netcool Process Agent as a non-privileged user (Windows) ... 159
Running the Netcool Process Agent as a privileged user ... 159

Appendix A. Initial requirements gathering checklist ... 161

Appendix B. nco_test_gateway.sh ... 164

Notices ... 169

Trademarks ... 171

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

About this publication

The purpose of this document is to be a reference to anyone deploying IBM Netcool/OMNIbus either on its own or as part of a deployment of IBM Netcool Operations Insight, and to help them to implement a solution using *standard* methods and techniques. If deployments are done in a standard way, engineers that must subsequently work on an installation done by somebody else will more easily understand what has been configured and how.

This document details recommended best practices for when installing and configuring IBM Netcool/OMNIbus 8.1. It is not designed to replace official product documentation; instead it augments the official product documentation and provides universally practiced standard methods and practices for deployment and configuration.

Installations are always dependent on customer requirements. With that said, the best practices contained in this document should always be adhered to whenever possible.

Note that this document is not intended as a training manual. It is a prerequisite that anyone making use of this document has completed appropriate IBM Netcool/OMNIbus 8.1 training.

Intended audience

This publication is intended as essential reading for all technical staff that are responsible for:

- Developing IBM Netcool/OMNIbus solutions;
- Installing and administering IBM Netcool/OMNIbus;
- Supporting IBM Netcool/OMNIbus.

What this publication contains

This publication contains the following sections:

- *Chapter 1 Introduction* on page 1:

  This chapter provides an overview of the document and background history regarding the *Event Service Framework* (ESF) and its contribution to the latest best practice standard multitier architecture configuration that ships with IBM Netcool/OMNIbus 8.1.

- *Chapter 2 Planning* on page 2:

  This chapter provides detailed information regarding the planning phase of an IBM Netcool/OMNIbus deployment including: background information on ObjectServer profiling and trigger statistics, an overview of the standard multitier architecture configuration, the purpose of the different tiers in a multitier environment, requirements gathering, solution sizing both in terms of software and hardware, where to implement custom functionality, architectural considerations, hardware considerations, solution delivery, the use of debug mode, debugging techniques, stress testing and a note on the need to remove temporary files and folders from deployed systems.

- *Chapter 3 Install* on page 41:

  This chapter points to the relevant places in the official product documentation where information can be found relating to product installation and deployment. Topics covered include: the IBM prerequisite scanner, a note on installing Netcool/OMNIbus WebGUI, and a note on the importance of updating the ObjectServer schema when upgrading.

- *Chapter 4 ObjectServers* on page 43:

  This chapter covers the following topics: how to configure certain key ObjectServer properties, contains information about the default triggers and how to write custom triggers, information about the default escalations, how procedures should be used, how to use indexing, how custom fields should be added, how users and groups should be managed, how to configure the IDUC port when connecting IBM Netcool/OMNIbus components through a firewall, a note about the new

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

HTTP and OSLC ports, event housekeeping, provides strategies for backing up the ObjectServer, event flood control, *Accelerated Event Notification* (AEN), provides a recommendation for the maximum numbers of events that Event Lists should be configured to display, and explains how to use administrative filters to set up support for multitenancy.

- *Chapter 5 Probes* on page 99:

  This chapter covers: Probe property file management, recommendations for how to set the generic Probe properties, gives best practices around the writing of Probe rules files, how to use the name-value pair (NVP) functionality, introduces the *Netcool Knowledge Library* (NcKL), explains how to convert MIB files to Probe rules files, points to the product documentation that explains how to implement mechanisms that will detect anomalous event rates, and provides some Probe rules "tips and tricks" that provide methods for performing certain tasks within Probe rules files.

- *Chapter 6 Gateways* on page 119:

  This chapter covers generic Gateway properties, Gateway configuration files, configuration file location and management, and provides a method for deploying cold stand-by Gateways.

- *Chapter 7 Anatomy of the standard multitier architecture configuration* on page 129:

  This chapter provides an in-depth analysis of the design rationale and features of the standard multitier architecture configuration; including a detailed description of each property and component.

- *Chapter 8 Proxy servers* on page 155:

  This chapter includes an overview of Proxy servers and their application.

- *Chapter 9 Firewall Bridge server* on page 156:

  This chapter includes an overview of the Firewall Bridge server and points to the official documentation for information about how it should be deployed including some use cases.

- *Chapter 10 The Netcool Process Agent and machine start-up* on page 157:

  This chapter provides an overview of the Netcool Process Agent including best practice tips on its configuration, host machine start-up configuration and security considerations.

Publications

This section lists publications in the IBM Netcool/OMNIbus library and related documents.

Your IBM Netcool/OMNIbus Best Practices library

The following documents are available in the IBM Netcool/OMNIbus Best Practices library:

- *IBM Netcool/OMNIbus 8.1 Best Practices Guide*

  This is the primary best practices IBM Netcool/OMNIbus document and includes information on all aspects of IBM Netcool/OMNIbus best practices including: requirements gathering, solution delivery, writing triggers, procedures, Probe rules, implementing automated housekeeping and flood protection mechanisms, and a detailed description of the standard multitier architecture configuration and all its parts.

- *IBM Netcool/OMNIbus Large scale and geographically distributed architectures*

  This document contains a best practice architecture design for deploying IBM Netcool/OMNIbus on a large scale where very high event numbers are anticipated; or if the IBM Netcool/OMNIbus deployment needs to span multiple geographic regions.

- *IBM DB2 High Availability for IBM Netcool products - Best Practices*

  A number of Netcool products use a DB2 database to store configuration data. This document provides guidance on a best practice method of implementing DB2 high availability using High

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Availability Disaster Recovery (HADR) thereby providing a resilient database service to the connecting Netcool application.

- *IBM Netcool/OMNIbus TEC Migration Best Practices*

  The purpose of this document is to be a reference to anyone integrating event flows from Tivoli Enterprise Console to IBM Netcool/OMNIbus using Tivoli EIF. This document details recommended best practices for when using the IBM Netcool/OMNIbus EIF probe and ITM LogFileAgent.

- *Upgrading IBM Netcool/OMNIbus in Production Environments Best Practices*

  This document provides practical tips for upgrading IBM Netcool/OMNIbus 7.x either *in situ* or where new hardware is deployed. The information contained in this publication is particularly pertinent in a production environment where data preservation is paramount, as is the need to minimise any kind of system outages. Note that although this document was originally written for Netcool/OMNIbus version 7.x, it is also equally applicable to Netcool/OMNIbus version 8.1.

All the publications listed above can be accessed online via the following URL:

http://ibm.biz/nco_bps

Note: In addition to the Netcool/OMNIbus core documents, best practices documents are also available for IBM Netcool Operations Insight from the same location.

Prerequisite publications

To use the information in this publication effectively, you must have some prerequisite knowledge, which you can obtain from the official product documentation listed below.

- *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*

  Includes installation and upgrade procedures for IBM Netcool/OMNIbus, and describes how to configure security and component communications. The publication also includes examples of IBM Netcool/OMNIbus architectures and describes how to implement them.

- *IBM Netcool/OMNIbus 8.1 Administration Guide*

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

- *IBM Netcool/OMNIbus 8.1 Probe and Gateway Guide*

  Contains introductory and reference information about Probes and Gateways, including probe rules file syntax and gateway commands.

All the publications listed above can be accessed online via the following URL:

http://ibm.biz/netcool_omnibus_pdf

Conventions used in this publication

This publication uses several conventions for special terms and actions and operating system-dependent commands and paths.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:** and **Operating system considerations:**)
- Keywords and parameters in text

Italic

- Citations (examples: titles of publications, diskettes, and CDs)
- Words defined in text (example: a nonswitched line is called a *point-to-point* line)
- Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."; letters as letters example: "The LUN address must start with the letter *L*.")
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data
- Variables and values you must provide: ... where *myname* represents....

`Monospace`

- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

Operating system-dependent variables and paths

This publication uses the UNIX convention for specifying environment variables and for directory notation.

When using the Windows command line, replace $variable with %variable% for environment variables, and replace each forward slash (/) with a backslash (\) in directory paths. For example, on UNIX systems, the $NCHOME environment variable specifies the directory where the IBM Netcool/OMNIbus core components are installed. On Windows systems, the same environment variable is %NCHOME%. The names of environment variables are not always the same in the Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX environments.

If you are using the bash shell on a Windows system, you can use the UNIX conventions.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Preface: The Event Service Framework (ESF)

Since its release, IBM Netcool/OMNIbus has been pushed further and further in terms of its event handling capacities. One of the many methods introduced to scale the product was the concept of *multitiered* IBM Netcool/OMNIbus architectures. By introducing a *Display layer*, IBM Netcool/OMNIbus systems could support many more users. By introducing a *Collection layer*, IBM Netcool/OMNIbus systems could support higher incoming event rates, as well as having a layer where event pre-processing can be carried out. By creating what was effectively a distributed database however (ie. the same data stored in multiple locations) the possibility of race conditions between ObjectServers was no trivial problem to solve, particularly before the release of IBM Netcool/OMNIbus version 7.x.

In response to this problem, the *Event Service Framework* (ESF) was devised. The ESF was essentially a set of "best practice" configurations for ObjectServers and Gateways which, when applied, would enable an engineer to rapidly deploy a multitiered IBM Netcool/OMNIbus architecture, without having to "reinvent the wheel" in terms of configuring event flow between the tiers. The ESF was a culmination of the best IBM Netcool/OMNIbus wisdom and knowledge of the day and drew input from Development, Tech Support and those that worked in the field. Since its creation, the ESF has been used extensively in customer deployments around the world.

Despite its widespread use however, the ESF was neither productised nor officially released. Prior to the release of IBM Netcool/OMNIbus 7.3.0 however, a project was initiated to update, streamline, and optimise the "standard" ESF model with a view to officially releasing it and shipping it with IBM Netcool/OMNIbus. This resulted in the new *standard multitier architecture configuration*, and is shipped with IBM Netcool/OMNIbus 7.3.0 onwards.

The standard multitier architecture configuration can be found under the `$OMNIHOME/extensions/multitier/` directory and is fully supported by IBM. It can be used to rapidly build out a one, two or three tiered IBM Netcool/OMNIbus architecture. In fact, it is recommended to use it in any deployment where more than one ObjectServer is deployed — for example: even for just a single failover pair, since it contains triggers and configuration to ensure failover and failback work correctly without event loss.

Note: When a single failover pair is deployed, it should be configured as an *Aggregation* pair. An Aggregation pair forms the basis of any IBM Netcool/OMNIbus architecture.

The standard multitier architecture configuration is referred to extensively throughout this document and should be used to build out any base IBM Netcool/OMNIbus architecture. The principles outlined in this document should then be followed to deploy custom configuration over the top. The resulting solution should then be simple, efficient, modular, and standardised, resulting in a more portable and more easily supportable and maintainable IBM Netcool/OMNIbus solution.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 1 Introduction

This document details the best practices that should be employed when deploying IBM Netcool/OMNIbus 8.1. This document is designed to help an engineer quickly and efficiently install a working system and ensure that if another engineer must subsequently work on a deployment installed by someone else, they will immediately understand what has been configured and how.

Many of the best practice concepts covered in this document have remain unchanged for many years and trace their origins to the *Netcool Certified Consultant* programme and previously published best practice guides. This document includes the relevant concepts from these sources and introduces new ones based on the new technologies and product features in the latest releases of IBM Netcool/OMNIbus.

Many of the best practice concepts are incorporated into the standard multitier architecture configuration hence it is referred to extensively throughout this document.

This document is intended to be read by anyone preparing to deploy or work with IBM Netcool/OMNIbus 8.1.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 2 Planning

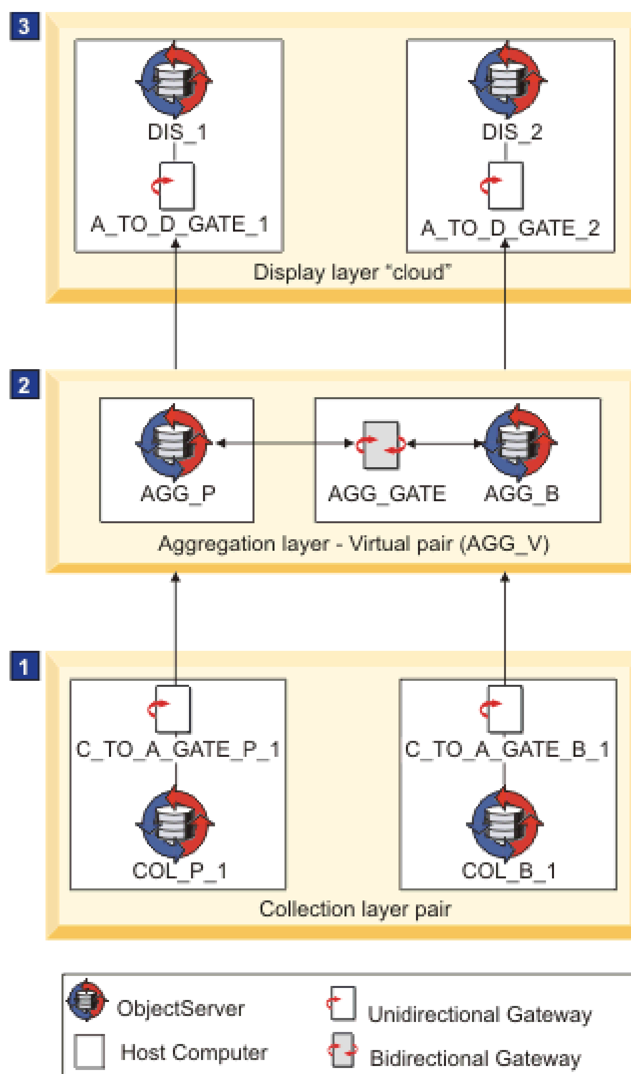
When architecting an IBM Netcool/OMNIbus solution, careful planning must be undertaken to ensure all necessary factors have been considered before provisioning both the software and hardware components, as well as designing the architecture layout.

This section lists the main factors that should be considered when designing an IBM Netcool/OMNIbus architecture and gives guidance on how to calculate reasonable initial component numbers.

Note: It is also strongly recommended to read *Chapter 4: Planning for installation or upgrade* and *Chapter 8: Configuring and deploying a multitiered architecture* in the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide* ahead of planning any IBM Netcool/OMNIbus deployment.

Standard Multitier Architecture Configuration

As mentioned in the preface of this document, the *Standard Multitier Architecture Configuration* (SMAC) is the pre-canned, best practice IBM Netcool/OMNIbus configuration that ships with the product and can be used to deploy one, two, or three tier systems.



© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The diagram above depicts a standard three tier deployment with a Collection ObjectServer layer (1), an Aggregation ObjectServer layer (2) and a Display ObjectServer layer (3).

The Aggregation layer represents the core of any IBM Netcool/OMNIbus deployment and can stand alone as a single tier system. The Collection and Display layers can be added if requirements or loading characteristics necessitate it.

It is recommended to use the standard multitier architecture configuration when deploying IBM Netcool/OMNIbus into *any* environment.

Note: More information on the standard multitier architecture configuration and how to deploy it can be found in: *Chapter 8: Configuring and deploying a multitiered architecture* in the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*.

A starting point

It is a general Netcool best practice to include resiliency in any solution. For example, rather than just installing a single standalone ObjectServer, it is better to install a failover pair of ObjectServers to provide resiliency in the event of an outage.

The starting point for any IBM Netcool/OMNIbus installation therefore is an Aggregation ObjectServer pair. Then, depending on the factors discussed in this section, other ObjectServers can be “bolted on” — namely: Display or Collection layer ObjectServers. Note that it may well be the case that neither a Collection layer nor a Display layer is needed. The factors discussed in this chapter will help an IBM Netcool deployment professional ascertain this.

Assuming the standard multitier architecture configuration has been used, even if a Display or Collection layer is not needed initially, either layer can be quickly and efficiently deployed at a later stage if circumstances change and an overall increased load necessitates it.

Background: Interpreting the ObjectServer profiling log

The information in the ObjectServer profiling log is a measure of the load that external clients are placing on the ObjectServer. Each granularity cycle of 60 seconds, the ObjectServer will write to the profiling log file a summary of the time it has spent interacting with each external client, a summary grouped by external client type and a total time for that granularity period, called “report period”.

The total time in a report period shows first how long the report period is (in brackets) followed by how much work the ObjectServer has managed to do in that period for all connected client processes. An indication of whether the ObjectServer is overloaded is if the report period is steady at 60 seconds or is frequently longer than 60 seconds. If it is greater than 60 seconds, it indicates that the ObjectServer has been too busy doing other things to write its report period summary to log, hence the report period value is greater than the granularity value of 60 seconds.

- Healthy system example:

```
Total time in the report period (60.0013883):  5.23s
```

The report period is 60 seconds, and the total time used is only 5.23 seconds. It is unlikely there will be any performance issues on this ObjectServer.

- Healthy system example:

```
Total time in the report period (60.0013883):  80.01s
```

The report period is 60 seconds, but the total time used is 80 seconds which shows that the ObjectServer has performed work in parallel. The ObjectServer can service read operations simultaneously and so it is quite possible to see such statistics in the ObjectServer profiling log file. It is possible but unlikely that there will be any performance issues on this ObjectServer.

- Unhealthy system example:

```
Total time in the report period (110.869982s):  99.0
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The report period is 110 seconds, but the ObjectServer has only managed 99 seconds of work dealing with external clients. This is an indication of an unhealthy ObjectServer if it is a frequent occurrence as it shows the ObjectServer was too busy to write profiling data at the end of its 60 second granularity cycle. In this example, the relatively high profiling figure of 99 seconds suggests that it is heavy client process load that is causing the overall overloading. Further analysis of the profiling log file will reveal who the heaviest users are.

- Unhealthy system example:

```
Total time in the report period (110.869982s):   1.1
```

The report period is 110 seconds, but the ObjectServer has only managed just over 1 second of work dealing with external clients. This is an indication of an unhealthy ObjectServer if it is a frequent occurrence as it shows the ObjectServer was too busy to write profiling data at the end of its 60 second granularity cycle. In this example, the total profiling time of 1 second suggests it is not the external client processes causing the load. The first place to investigate therefore would be the trigger statistics log file as the overloading is likely being caused by a poorly performing ObjectServer trigger or triggers.

Note: The ObjectServer profiling log file can be found in the `$OMNIHOME/log/` directory. For example, the current profiling log for the ObjectServer AGG_P will be called: `AGG_P_profiler_report.log1`.

Background: Interpreting the ObjectServer trigger statistics log

If the ObjectServer profiling log shows that the report period is frequently greater than 60 seconds indicating it is overloaded, yet the amount of time the ObjectServer spent interacting with external clients is relatively low, the cause may be due to a poorly performing ObjectServer trigger. In such circumstances, the trigger statistics log file should be examined as it is likely a poorly performing trigger or multiple triggers are causing the overloading. In addition to routinely monitoring the profiling log of the ObjectServer therefore, the trigger statistics log file should similarly be routinely monitored.

Note: Netcool/OMNIbus 8.1 comes pre-installed with self-monitoring automations that automatically monitor profiling and trigger statistics information, among other things. Should there be a breach of any pre-defined thresholds, a synthetic alarm will be generated and can be seen on the WebGUI *Netcool Health* dashboard that is provided out of the box.

The information in the ObjectServer trigger statistics log is an exact measure of the load that ObjectServer triggers are placing on the ObjectServer. Each granularity cycle of 60 seconds, the ObjectServer will write to the trigger statistics log file the amount of time each trigger has consumed within that granularity period, plus an overall sum for all triggers for that granularity period.

In the profiling log file, it is possible that operations performed by external clients represented in the log file can overlap. This is because the ObjectServer supports read operations running in parallel. Examples of this are the reader components of connected Gateways. When triggers fire however, they execute one at a time in a single threaded fashion to ensure data integrity. Hence if the total trigger time is consistently greater than the report period as in the above example, this is an indication that the ObjectServer is getting further and further behind in its tasks.

Note: While an ObjectServer trigger is firing, the ObjectServer's entire data store is exclusively locked to other write operations for the duration of the trigger execution. Read operations however can run concurrently and are only blocked momentarily on reading a particular row if the active write thread is modifying that same row at the same moment. This data store locking model is a recent addition to IBM Netcool/OMNIbus (7.3.0 onwards) and is a lot less restrictive than previous versions.

The following is an example excerpt from a trigger statistics log file:

```
Trigger Group 'widgetcom_triggers'
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
Trigger time for 'bad_trigger': 66.049248s
...
Time for all triggers in report period (60s): 86.737066s
```

The above trigger statistics log file excerpt shows that the ObjectServer was behind in its tasks for this granularity period since it took 86 seconds to run all its triggers; meaning that it was behind by 26 seconds. Further analysis of the trigger statistics log file shows that the trigger called “bad_trigger” is likely the primary culprit for this high trigger statistics total.

Note: Unlike the profiling log, the report period in the trigger statistics log will always show the report period as 60 seconds, which is the granularity setting for the ObjectServer.

Any trigger identified in the trigger statistics log as a poor performer should be urgently reviewed to see why it is performing so poorly so that remedial action may be taken. In a production environment, initial triage actions may include the disabling of the suspect trigger, so as not to cause an outage of the system.

Note: The ObjectServer trigger statistics log file can be found in the `$OMNIHOME/log/` directory. For example, the current trigger statistics log for the ObjectServer AGG_P will be called: `AGG_P_trigger_stats.log1`.

Multitiered deployment: the purpose of each tier

There are a number of scenarios described in this chapter that illustrate when and why a Collection or Display layer may need to be added to an IBM Netcool/OMNIbus deployment and thus transform a single tiered IBM Netcool/OMNIbus system into a multitiered one. Typically, the scenarios discussed either relate to taking load off the Aggregation layer, or where other architectural or network constraints necessitate additional tiers.

The load on an ObjectServer can be established by examining both the profiling log and the trigger stats log:

- The profiling log provides a summary for each ObjectServer granularity cycle of how much time the ObjectServer spent interacting with processes *external* to the ObjectServer.
- The trigger statistics log provides a summary for each ObjectServer granularity cycle of how much time the ObjectServer spent running *internal* processes — specifically: triggers.

The average “total” time taken from both the profiling and trigger statistics logs can be added together to give an indication of ObjectServer load. If the combined total is *approaching* the ObjectServer granularity time of 60 seconds, the profiling and trigger statistics logs should be analysed to ascertain what the biggest users of the ObjectServer’s time are; with a view to taking remedial action in order to reduce the overall total.

For example, remedial actions may include:

- Making an inefficient trigger identified in the trigger statistics log more efficient;

Note: See the section entitled: *Custom triggers* later in this document for more information on how to create well written, efficient triggers.

- Streamlining IBM Netcool/Impact policies to minimise ObjectServer read and write operations; thereby reducing the time in the profiling log that the ObjectServer spends interacting with IBM Netcool/Impact;
- Deploying a Display ObjectServer layer where user client loading is identified in the profiling log as a high consumer.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: IBM Netcool/OMNIbus 8.1 is extremely efficient for read operations. It is therefore recommended not to use Display ObjectServers to remedy performance issues except under the most extreme circumstances, where it can be proven that read load is causing performance issues.

- Deploying a Collection ObjectServer where Probe loading is identified in the profiling log as a relatively high consumer;

Note: Any remedial actions should be tested thoroughly before deploying into a production environment.

This section briefly outlines the purpose of each tier in terms of the capabilities each provides and gives guidance as to when each tier might be deployed to either reduce load on the Aggregation layer or to resolve an architectural challenge where, for example, firewalls are present within the deployment architecture.

AGGREGATION LAYER

The Aggregation tier or layer is the heart of any IBM Netcool/OMNIbus deployment and is where all events from all sources are aggregated together into one place, hence its name. The following list summarises the main capabilities that an Aggregation layer provides:

- *Event receipt*

  The Aggregation layer is configured to receive events directly from Probes, from Collection to Aggregation ObjectServer Gateways, from the Aggregation failover Gateway or any other external source. ObjectServer Gateways insert events into the ObjectServer in batches whereas Probes insert events either individually or in very small batches. Probes therefore typically induce greater load on ObjectServers than Gateways do. Probes are also typically the primary source of events in any IBM Netcool/OMNIbus deployment. Hence Probes typically make up the bulk of the induced ObjectServer load under this category.

- *Event correlation and processing*

  Since the Aggregation layer is where all events are brought together into one place, it is the natural point where event correlation and general event processing should be carried out. Events may be correlated against each other, they may be enriched, or they may transition through different states as they are processed multiple times by automated processes or acted on by operators via Event List tools.

  The Aggregation layer is where the bulk of custom triggers will normally be installed since this is the primary place where events are manipulated and managed.

Note: At the Collection layer, events are forwarded to the Aggregation layer and then deleted. Because the Collection layer does not retain events after they have been forwarded, it is typically not feasible to perform reliable event correlation at the Collection layer. One-time event processing is certainly possible at the Collection layer, however, before events are forwarded up to the Aggregation layer.

- *Connection point for integrations*

  Since the Aggregation layer holds the “master copy” of the event set, it is normally the primary point of connection for ticketing Gateways, database Gateways, such as the JDBC Gateway or the Oracle Gateway, or any other sort of integration — for example: IBM Netcool/Impact.

- *Servicing of user displays*

  IBM Netcool/OMNIbus WebGUI or Native Event List clients may connect directly to the Aggregation layer for the purpose of viewing and working the events contained therein.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: The ObjectServer is multi-threaded for read operations and so can service read requests from multiple connected read-clients simultaneously. The profiling logs will give only an *indication* therefore of ObjectServer load and is not a definitive measure. User dashboard performance can also be evaluated anecdotally, by speaking with users and getting feedback on Event List responsiveness, for example. See the sections entitled: *Background: Interpreting the ObjectServer* profiling log and *Background: Interpreting the ObjectServer trigger* statistics log later in this chapter for more information on how to determine the load on an ObjectServer.

COLLECTION LAYER

One or more Collection layer pairs of ObjectServers can be added below an existing Aggregation pair to provide greater scalability and to provide additional capabilities. The primary purpose of a Collection layer is for the collection or receipt of the incoming events. Additional optional functions performed by Collection layer ObjectServers are:

- One-time event enrichment;
- Filtering, suppression, and re-routing of events to, for example, an archive database;
- Automated event flood processing.

Note: See the section entitled *Event flood control* in *Chapter 4* for more discussion around event flood control mechanisms in the context of a Collection layer ObjectServer.

The following list summarises the main capabilities that a Collection layer provides:

- *Event receipt*

  If the ObjectServer profiling logs show that the Probes are collectively inducing a significant amount of load on the Aggregation layer, consideration should be given to installing a Collection layer.

  Installing a Collection layer can dramatically reduce the overall profiling time of the Aggregation ObjectServer by transferring the write load of the Probes to the Collection layer. This helps in two main ways.

  First, Collection ObjectServers absorb much of the load imposed by multiple duplicate event insertions. In any granularity cycle, updates due to a deduplicated event will be presented to the Aggregation layer ObjectServer via the Gateway as a single event record; thus eliminating the load of many event inserts or updates of the same event.

  Second, whereas Probes insert each occurrence of each event individually and induce load on the Aggregation ObjectServer for each one, a Collection to Aggregation ObjectServer Gateway instead inserts events into the Aggregation layer in batches, which is more efficient.

  Hence by removing Probes from the Aggregation layer and instead connecting the Probes to a Collection layer, the event data will be fed more efficiently into the Aggregation layer.

- *One-time event enrichment and processing*

  The Collection layer receives events from event sources, holds them until the Collection to Aggregation Gateway transfers them to the Aggregation layer, holds them again for a while longer, and then deletes them. The Collection layer effectively provides a marshalling area for events to arrive, be deduplicated and then sent up to the Aggregation layer.

  Once events have been forwarded up to the Aggregation layer, they are only retained at the Collection layer for a short time, and then they expire and are deleted. This process is necessary because the connection from Collection to Aggregation is unidirectional only, hence events need to be reaped from the Collection layer independently of their deletion at the Aggregation layer. Events are retained at the Collection layer for a time however, in case the event recurs. If the event does recur, deduplication occurs and the update is again forwarded up to the Aggregation layer with the updated event information.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Since the events in the Collection layer are relatively transient, correlation type activities cannot feasibly be carried out at this point. Correlation functions therefore should be carried out on the Aggregation layer.

One-time enrichment functions, however — for example: ones that are not dependent on the presence of other events — may be carried out at the Collection layer. Functions such as these may include triggers that perform enrichment of events based on information stored in custom tables; or enrichment via IBM Netcool/Impact policies.

In the standard multitier architecture configuration, the Collection to Aggregation Gateways are filtered to read events where the field `SentToAgg` is 0 — ie: the event has not yet been sent to the Aggregation layer. After sending each, the Gateway sets this field to 1 so that it is not subsequently re-sent to the Aggregation layer when it is not supposed to. The deduplication trigger is designed to reset the field `SentToAgg` back to 0 each time the event recurs.

If one-time Collection layer enrichment is required, a custom trigger can be created to set the `SentToAgg` to 2, for example, so that it can be first processed by any functions either present or connected to the Collection layer. After processing, the `SentToAgg` can be set to 0 and forwarded to the Aggregation layer in the normal way.

EXAMPLE:

Widgetcom have some one-time event enrichment functionality they wish IBM Netcool/Impact to carry out on all new events prior to sending to the Aggregation layer.

The Netcool Administrator creates the following trigger to be added to the Collection layer *solution delivery SQL file*:

```
-------------------------------------------------------------------
-- CREATE A TRIGGER TO FLAG EVENTS FOR ONE-TIME ENRICHMENT
CREATE OR REPLACE TRIGGER flag_for_enrichment
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER flag_for_enrichment
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2012
This trigger flags all incoming inserts for enrichment by Tivoli
Netcool/Impact by setting SentToAgg to 2.
After enrichment, SentToAgg will be set to 0 and sent up to the
Aggregation layer in the normal way.'
BEFORE INSERT ON alerts.status
FOR EACH ROW
begin

   -- FLAG EVENT FOR ENRICHMENT
   set new.SentToAgg = 2;

end;
go
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Netcool Administrator takes care to ensure that the IBM Netcool/Impact policy responsible for the enrichment of the event also updates the `SentToAgg` field to 0 so that the event will be picked up by the Collection to Aggregation Gateway and forwarded to the Aggregation layer.

Note: See the section entitled: *Solution delivery* for a more thorough explanation of what *solution delivery SQL files* are and their purpose. Also see the section entitled: *The Collection layer* in *Chapter 7: Anatomy of the standard multitier architecture configuration* for a more detailed description of the way in which the standard multitier architecture configuration controls the flow of events between the Collection and Aggregation layers.

- *Connection point for event archiving*

  In many cases, certain groups of events need to be retained in the event archive, but operators do not need to see them; nor are they required for any correlation type functionality. Such events are ideal candidates for directly archiving from and subsequently deleting from the Collection layer.

Note: The term *event archiving* in this context means the sending of events to an event archive such as a DB2 or Oracle database via an IBM Netcool Gateway.

As in the previous example, the `SentToAgg` flag can be used to selectively flag events for archiving. Once archived, they can be marked for deletion in the same way that the Collection to Aggregation ObjectServer Gateways do: by setting the the `SentToAgg` flag to 1. The Collection expire trigger will then delete the event up to 90 seconds later.

EXAMPLE:

Widgetcom have a requirement that events with event code 12 need to be saved to the event archive via the JDBC Gateway that sends events out to the DB2 event archive database. These events do not need to be displayed to operators nor are they required for any correlation type functionality.

The Netcool Administrator creates the following database trigger to be added to the Collection layer *solution delivery SQL file* that will flag event code 12 events for archiving:

```
------------------------------------------------------------------
-- CREATE A TRIGGER TO FLAG INSERTS FOR ARCHIVING
CREATE OR REPLACE TRIGGER flag_for_archive
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER flag_for_archive
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2012
This trigger acts only on events with event code 12.  Events of this
type are flagged for archiving via the DB2 Gateway and are not
passed up to the Aggregation layer.  The DB2 Gateway table
replication file flags events for deletion after they have been sent
to the event archive.'
BEFORE INSERT ON alerts.status
FOR EACH ROW
WHEN new.EventType = 12
begin
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
   -- FLAG EVENT FOR ARCHIVING
   set new.SentToAgg = 3;
end;
go
```

To ensure that reinserts are also processed in the same way, the Netcool Administrator creates a second database trigger to handle reinserts for event code 12 events and again adds it to the Collection layer *solution delivery SQL file*:

```
-------------------------------------------------------------------
-- CREATE A TRIGGER TO FLAG REINSERTS FOR ARCHIVING
CREATE OR REPLACE TRIGGER flag_for_archive
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER flag_for_archive
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2012
This trigger acts only on events with event code 12.  Events of this
type are flagged for archiving via the DB2 Gateway and are not
passed up to the Aggregation layer.  The DB2 Gateway table
replication file flags events for deletion after they have been sent
to the event archive.'
BEFORE REINSERT ON alerts.status
FOR EACH ROW
WHEN new.EventType = 12
begin

   -- FLAG EVENT FOR ARCHIVING
   set new.SentToAgg = 3;
end;
go
```

The Netcool Administrator takes care to ensure the JDBC Gateway table replication file is configured so that it is filtered to only reads events where the `SentToAgg` field is set to 3; and also, that the Gateway then sets the `SentToAgg` field to 1 after reading the event — ie: after IDUC — so that the event will be expired in the normal way.

Note: See the section entitled: *Solution delivery* for a more thorough explanation of what *solution delivery SQL files* are and their purpose. Also see the section entitled: *The Collection layer* in *Chapter 7: Anatomy of the standard multitier architecture configuration* for a more detailed description of the way in which the standard multitier architecture configuration controls the flow of events between the Collection and Aggregation layers.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- *Event flood auto-protection mechanisms*

  The Collection layer can be used to implement a variety of automated event flood protection mechanisms to help prevent the Aggregation layer from being overwhelmed. For example, the Collection layer might have triggers that monitor event numbers or rates — and delete or divert events to overflow ObjectServers from the Collection layer if a threshold is breached.

  EXAMPLE:

  Widgetcom have a requirement that if the event rate of unique events received at any one Collection ObjectServer exceeds 1,000 in the last 60 seconds, then all events that are lower than Major severity should be deleted.

  The Netcool Administrator creates the following temporal trigger to be added to the Collection layer *solution delivery SQL file* that will delete all events lower than Major severity that have not already been sent to the Aggregation layer.

```
------------------------------------------------------------------
-- CREATE A TRIGGER TO DETECT EVENT FLOODS AND TAKE REMEDIAL ACTION
CREATE OR REPLACE TRIGGER delete_flood_events
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER delete_flood_events
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2012
This trigger runs once every 31 seconds and counts the number of
events received in the last 60 seconds.  If the number exceeds 1000
inserts or reinserts, all events less than Major severity will be
deleted.'
EVERY 31 SECONDS
declare
   runningtotal integer;

begin
   -- INITIALISE VARIABLE
   set runningtotal = 0;

   -- COUNT UP ALL UNIQUE EVENTS THAT HAVE OCCURRED WITHIN THE LAST
   -- 60 SECONDS BUT HAVE NOT YET BEEN SENT TO THE AGGREGATION LAYER
   for each row thisrow in alerts.status where
          thisrow.SentToAgg = 0 and
          thisrow.LastOccurrence > (getdate() - 60)
   begin

          -- INCREMENT THE COUNTER
          set runningtotal = runningtotal + 1;
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
   end;

   -- IF THE TOTAL IS MORE THAN 1000 UNIQUE EVENTS
   if (runningtotal > 1000) then

         -- DELETE ALL EVENTS LESS THAN MAJOR SEVERITY
         delete from alerts.status where Severity < 3;
   end if;
end;
go
```

In this example, the temporal trigger is set to run once every 31 seconds and will examine the number of unique events that have been received within the last 60 second period from that moment in time.

It could have been implemented as a database trigger and set to perform a count each time a new event is inserted, however such a trigger may impose unnecessary load on the ObjectServer since the trigger code incurs a scan of the *alerts.status* table each time it counts up the number of recently inserted rows.

Note: See the section entitled *Custom triggers* in *Chapter 4: ObjectServers* for more information about the different types of ObjectServer triggers, when each type should be considered and how to build them.

- *Network restrictions and limitations between Probes and the Aggregation layer*

  It is a common scenario that the Aggregation ObjectServer pair is located in a network operations centre on a "core" network; meanwhile the Probes that collect events are located close to the managed entities that generate the events.

  In some cases, there may be a firewall between the Probes and the Aggregation ObjectServers and, for security reasons, components outside of the firewall are not permitted to make connections from outside the core network to processes running inside the core network. In this case, it is possible to deploy a Collection ObjectServer layer outside the core network and have Probes connect to that instead. The Gateway *process* that connects the Collection layer to the Aggregation layer can be located on the core network and connect out to the Collection ObjectServer layer. Event flow will still travel from Collection to Aggregation, it is just the initial connection that will be made from inside the core network to outside the core network.

Note: Another option in this scenario would be to use the Netcool Firewall Bridge. See the section entitled: *Firewall Bridge server* later in this document for more information.

In other cases, there may be limited bandwidth between the network where the managed entities are located and the Aggregation ObjectServer pair — for example: over a wide area network (WAN). If the event set has a lot of deduplication, it can reduce network traffic by having events deduplicate on a Collection layer ObjectServer before being forwarded over the WAN to the Aggregation pair. Also, automation can be designed to reside on the Collection layer ObjectServers to reduce and consolidate the events going up to the Aggregation layer thereby further reducing the WAN traffic.

DISPLAY LAYER

A Display layer of ObjectServers should be considered if the following apply:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- The primary Aggregation ObjectServer profiling log shows that Native Event List clients or WEBTOP are consuming very high amounts of ObjectServer profiling time (eg. 100+ seconds) resulting in the report period being pushed beyond 60 seconds;
- Users are reporting sluggish refreshes in their Event Lists or dashboard views.

Note: Since IBM Netcool/OMNIbus 7.4, there has been a dramatic improvement in transactional concurrency in the ObjectServer for read operations. This means that it is unlikely that most deployments of modern versions of Netcool/OMNIbus will need an ObjectServer Display layer to resolve client-induced load performance issues.

The Aggregation layer is the central point where events from all sources are aggregated together and stored. It is the point where event correlation can be done, general event processing and manipulation is done and is the primary point where third-party integrations are done.

In the same way that the Collection layer can be installed to alleviate load off the Aggregation layer, so too can a Display layer of ObjectServers be deployed to ease the load of supporting user clients connecting to the Aggregation layer.

Note: In terms of initial provisioning, one Display ObjectServer should be provisioned per 100 users plus one additional Display ObjectServer for resiliency — ie: N + 1. For example, a system designed to support 200 concurrent users should have: 200 users/100 users per Display ObjectServer = 2 Display ObjectServers + 1 for resiliency — giving 3 Display ObjectServers in total. These are initial provisioning guidelines and may be affected by additional factors. See the section entitled: *Estimated number of users* later in this chapter for more discussion on end user induced load, and the factors around when a Display layer should be considered.

The following list summarises the main capabilities that a Display layer provides:

- *Event List and dashboard support*

  The primary purpose of the Display layer is to support Native Event List or web-based dashboard clients. Instead of the Aggregation ObjectServer incurring the load of servicing the requests from these clients, the Display layer provides a dedicated point through which these requests can be proxied.

  The Display layer has only a minimal set of triggers enabled since no event processing occurs at the at the Display layer. Typically, the only write operations to events at the Display layer occur when either a user executes a tool or the Aggregation to Display ObjectServer Gateway performs an IDUC refresh from the Aggregation layer.

  Since write operations are minimised within Display layer ObjectServers, they are highly available for servicing client read requests — for example: when an Event List refreshes. The ObjectServer is capable of processing multiple read requests simultaneously. Hence with a Display layer deployed, it can enhance the end user experience in terms of refresh responsiveness.

- *Read-only event source for third-party integrations*

  All Display layer ObjectServers are essentially a replica of the Aggregation layer ObjectServers and contain a near real-time copy of the event data. If a particular deployment is to have a high-load, third-party integration attached, that needs a read-only feed of the events, a Display layer ObjectServer can provide this.

  Instead of connecting the third-party client directly to the Aggregation layer where it might impose additional load, the client could instead be connected to a Display ObjectServer which can relay the same information; albeit slightly delayed due to the inter-tier Gateways.

  To ensure resiliency, the third-party client should have at least two Display ObjectServers listed in its configuration: one as a primary source, the other as a backup source.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: This section outlines the purposes of each tier and under what circumstances one might consider the deployment of a Collection or Display layer. Also see the section entitled: *Provisioning guidelines summary* later in this document which outlines general guidance on when these additional tiers might be considered during the initial design phase.

Use the standard multitier architecture configuration

Even if Collection or Display ObjectServers are not going to be used in a solution, the pair of ObjectServers that are deployed should still be configured as an Aggregation ObjectServer pair using the standard multitier ObjectServer and Gateway configuration provided with the product, regardless of how many tiers will initially be deployed. Not only does this ensure failover and failback will work correctly, it also means that Collection or Display ObjectServers can be added to the solution very easily in the future if required, since the necessary trigger configuration on the Aggregation ObjectServers will already be in-place to accommodate the additional components.

Note: The Standard Multitier Architecture Configuration (SMAC) is shipped with IBM Netcool/OMNIbus since version 7.3.0 and can be found in `$OMNIHOME/extensions/multitier/`. Detailed instructions on how to deploy it can be found in *Chapter 8: Configuring and deploying a multitiered architecture* of the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*.

Use the initial configuration wizard

Once the numbers and locations of components has been deduced, it may be convenient to use the Netcool/OMNIbus Initial Configuration Wizard (ICW) to deploy the ObjectServer and ObjectServer Gateway components on the target boxes. Not only does this speed up the deployment process, it also ensures a consistent approach to the deployment and reduces the chance of any typos or other errors from being made as ObjectServers are being built and configuration files being copied.

Note: The Netcool/OMNIbus ICW uses the Standard Multitier Architecture Configuration as well as the best practice naming conventions by default when it deploys ObjectServers and ObjectServer Gateways.

The Netcool/OMNIbus ICW is run via the command: `$OMNIHOME/bin/nco_icw`.

Further information regarding the Netcool/OMNIbus ICW can be found on the IBM Knowledge Center:

http://www.ibm.com/support/knowledgecenter/SSSHTQ_8.1.0/com.ibm.netcool_OMNIbus.doc_8.1.0/omnibus/wip/install/reference/omn_ins_ict.html

Note: If the Netcool/OMNIbus ICW is used, the resulting deployment descriptor file should be included in the deployment deliverable set. See the section entitled: *Solution delivery* for more information on how to deliver the overall solution configuration.

Requirements gathering

The starting point of any Netcool architecture deployment is requirements gathering. Below is a non-exhaustive list of some requirement metrics that should be gathered prior to designing an IBM Netcool architecture or planning an IBM Netcool/OMNIbus deployment.

Note that the factors included in this section need to be considered collectively when initially sizing a solution. Any design resulting from these initial metrics will constitute a starting point for a solution, and may need to change when additional components that add load are added.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The initial requirement metrics that should be gathered are:

- An estimate of the total number of managed entities that events will come from.
- An estimate of the total number of events the Probes will need to process from the event sources per day (ie. event rate), including an estimate of the *peak* event rate. System design should always be built to manage peak rates plus a contingency factor.
- Based on the event rate and the event types (eg. syslog, SNMP traps, etc.), an estimate of the number of Probes that will be required to receive and handle the event load. The maximum event throughput for Probes varies on many factors — including: Probe type; whether the Probe is multi-threaded or single-threaded, if the Probe incurs file I/O (eg. Syslog Probe vs. SNMP Probe), or the hardware the Probe is running on. A suggested conservative estimate for the purposes of initial architecture design is 100 events per second for single-threaded Probes (eg. Syslog Probe) and 200 events per second for multi-threaded Probes (eg. SNMP Probe).

Note: See the section entitled *Provisioning guidelines summary* for more guidance on the provisioning of Probes and other components.

- An estimate of the event rate that will subsequently be sent to the ObjectServers from all Probes, after any event discards have been done at the Probe level.
- An estimate of how many events will deduplicate, and therefore an estimate of the maximum number of standing rows in the ObjectServer at any one time.
- An estimate of the maximum number of concurrent users of the system.
- A high-level outline of the event expiry policy that will be implemented. This is important to ensure events don't remain in the ObjectServer indefinitely. See the section entitled *ExpireTime* later in this document for examples of how to formulate an event expiry policy.
- A list of all components that will connect or interact with the ObjectServer (ie. and thereby potentially impose ObjectServer load). This includes other IBM Netcool components.
- A list of all custom functionality required within the deployment. Ideally each piece of custom functionality should be noted down in pseudocode.

A sample checklist is provided in *Appendix A* at the end of this document.

Solution sizing and design based on requirements

The following subsections provide background into some of the metrics outlined in the previous section of this document entitled *Requirements gathering* and describe each metric’s significance to overall performance.

Estimating the number of managed entities

The number of devices or managed entities to be monitored will give an indication as to the number of events the Netcool system will receive and help to determine expected load as well as how many and what type of Probes will be required.

The type of devices or managed entities must be considered because some types of event source are more “chatty” than others. For example, a router will likely generate more events than a server.

EXAMPLE:

Widgetcom have 1,000 Cisco switches and a combination of 5,000 servers and workstations that will be managed by Netcool. The number of managed entities in this case is 6,000 therefore.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Estimated number of events per day and event rate

Based on the number of managed entities, their types, and their configuration, we can estimate the number of events we can expect to receive each day and thereby calculate an average event rate.

EXAMPLE:

Following on from the previous example, *Widgetcom* have 1,000 Cisco switches in their environment that send approximately 5,000 traps per day each to the Netcool system. In addition to this, they have 5,000 servers and workstations; each sending approximately 100 syslog messages per day to Netcool.

This amounts to 5,000,000 traps per day hitting Netcool and 500,000 syslog messages per day which translates into a combined event rate of: (5,500,000 events per day ÷ (24 × 60 × 60) seconds per day) = 64 events per second potentially hitting the ObjectServer infrastructure.

Estimated number of Probes

Based on the event source types, we can identify the types of Probes that will need to be deployed. Based on the expected event rate for each event source type, we can deduce the number of each type of Probe that will be required.

Note: See the section entitled: *Provisioning guidelines summary* later in this chapter for details on how to initially scope out the scale of an IBM Netcool deployment in terms of numbers of components based on the input metrics.

EXAMPLE:

Following on from the previous example, the incoming event data is expected to be 5,000,000 traps per day and 500,000 syslog messages.

An IBM Netcool SNMP Probe will handle the traps. 5,000,000 traps per day translates into: (5,000,000 events per day ÷ (24 × 60 × 60) seconds per day) = 58 events per second hitting the Probe. Since one Probe should be provisioned to handle up to 100 events per second, then one SNMP Probe should be sufficient. Note that consideration should be given to deploying a second SNMP Probe in peer-to-peer mode for resiliency.

The monitored servers and workstations will be configured to send their syslog messages to a syslog daemon Probe. 500,000 syslog messages per day translates into: (500,000 events per day ÷ (24 × 60 × 60) seconds per day) = 6 events per second hitting the Probe. Since one Probe should be provisioned to handle up to 100 events per second, one syslog daemon Probe should be sufficient. As for the SNMP Probe, consideration should be given to deploying a second syslog daemon Probe in peer-to-peer mode for resiliency.

Assessing Probe load on the ObjectServer

Each event that is inserted by a Probe places load on the ObjectServer. The overall load each Probe puts on the ObjectServer can be monitored via the ObjectServer profiling log.

The total time spent interacting with Probes will show up something like the following:

```
Execution time for all connections whose application name is
'PROBE': 30.048472s
```

The profiling log also contains a total for all client activity and looks like the following example:

```
Total time in the report period (80.002960s): 50.056615s
```

In the above example, the total time the ObjectServer spent interacting with external clients was just over 50 seconds, however the report period was 80 seconds which exceeds the ObjectServer's granularity value of 60 seconds. This is an indication that the ObjectServer is overloaded and performance *may* be degraded as a result if this occurs frequently.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This overloading may manifest itself in many ways including: a sluggish response from Event Lists connected to the ObjectServer; or delays in events being presented to operators. In the scenario above where Probes are showing as the major load, installing Collection layer ObjectServers should be considered to relieve the Aggregation layer of Probe-induced load.

While installing a Collection layer to alleviate Probe load may help ease the overall loading on the ObjectServer, other factors may also be at work. Hence checking the trigger statistics log is also important when evaluating ObjectServer loading, as there may be a poorly performing trigger, for example, that is contributing to the overloading.

The overall amount of load on the ObjectServer is a *combination* of the total time taken interacting with connected clients in any given report period (as per the profiling log data) with the total time used by internal triggers for the same period (as per the trigger stats log). The two totals need to be considered together to give an indication of overall load.

Note: See the sections entitled: *Background: Interpreting the ObjectServer* profiling log and *Background: Interpreting the ObjectServer trigger* statistics log earlier in this document for more information on understanding the profiling and trigger statistics logs.

Consideration for event bursts

An additional consideration is if bursts of events are occasionally received. In the previous example, the average event may well be 64 events per second, calculated across a day. There may be, for example, bursts of 1,000 events per second sustained for up to 10 seconds at a time, which an average will not reflect. Consideration therefore also needs to be given to as to whether buffering alone on the Probe will be sufficient to handle these bursts, or if additional Probes should be deployed and the load split across multiple Probes.

The maximum number of events a Probe can handle varies for each type of Probe. For example, some Probes are single-threaded, and some are multi-threaded. Additionally, the hardware characteristics and platform the Probe is running on can significantly affect its event throughput handling ability.

One option for handling event bursts at the Probe level is to consider putting in place event storm contingencies. For example, a Probe could be configured to start discarding low severity events if the event rate breaches a threshold.

Note: See the section entitled: *Detecting event floods and anomalous event rates* for more details on implementing event storm contingencies at the Probe level.

Estimated number of standing events in the ObjectServer

Not only is insertion rate a significant consideration with respect to ObjectServer load, so too is the number of standing events in the ObjectServer at any one time. The term *standing events* refers to the number of unique rows in the *alerts.status* table.

The number of events in the ObjectServer may influence heavily the amount of time a trigger takes to execute, particularly where a trigger is performing a scan of the event table *alerts.status*. Similarly, the number of rows in *any* table in the ObjectServer will impact on the amount of time a trigger takes to execute a scan of that table, and therefore impact the load on the ObjectServer.

Note: While an ObjectServer trigger is firing, the ObjectServer's entire data store is exclusively locked to other write operations for the duration of the trigger execution. Read operations however are allowed to run concurrently and are only blocked momentarily on reading a particular row if the active write thread is modifying that same row at the same moment. This data store locking model is a recent addition to IBM Netcool/OMNIbus (7.3.0 onwards) and is a lot less restrictive than previous versions.

Deduplication means that there is not a one-to-one mapping of events being inserted by Probes and the total count of events in the ObjectServer, since the same event recurring simply causes the Tally of an

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

existing event to increment. Additionally, triggers may carry out event correlation and deletion, reducing the total count of events further still.

Since event count so heavily impacts on profiling time and hence ObjectServer loading, it is an important consideration in the sizing of a solution. One of the goals of any solution design is to reduce the number of standing rows to those that either operators need to see or those that need to be present to participate in correlation or automation.

As a general guideline in terms of event numbers, a “small” Netcool system will have up to 10,000 standing rows at any time, a “medium” Netcool system will have between 10,000 and 50,000 and a “large” Netcool system anything more than that.

Ideally an IBM Netcool/OMNIbus ObjectServer will not contain more than 100,000 standing rows at any one time and should be streamlined to contain only the events that are needed to resolve current, outstanding issues. Once problems are resolved, the corresponding events should be deleted from IBM Netcool/OMNIbus. The ObjectServer is not intended to be an event archiving point. If events need to be retained for auditing purposes, they can be copied via an IBM Netcool Gateway to a historical event archive database — for example: a DB2 IBM Netcool REPORTER database.

When streamlining the event set:

- Consider how many events your operators can feasibly process in a day;
- Limit the number of events to those that your operations staff can process, and those that will participate in an automatic process — for example: events that will raise a ticket, participate in a correlation, or provide an indication of the health of a service;
- Aggressively archive and delete events that do not fall into the above categories or ones that relate to problems that are now resolved.

Once all custom functionality is in-place, an IBM Netcool/OMNIbus system must be stress tested up to the maximum number of events it is envisaged the ObjectServers are potentially going to have during peaks to ensure that the system can successfully process the load.

Note: It is advisable to engineer the end system to be able to handle 20% more events than the maximum number expected. This will give the system an amount of *"headroom"* in case there is a sudden, unexpected influx of events — for example: during an event storm.

As noted in the previous section, the amount of load on the ObjectServer is a combination of the time taken interacting with connected clients (as per the profiling log data) plus the total time used by internal triggers (as per the trigger stats log). If this total is approaching the granularity value, steps should be considered to extend the design to support the increasing load. This can be done a number of ways including: adding Collection and/or Display ObjectServers; or splitting the load across two separate IBM Netcool/OMNIbus systems.

Note: IBM Netcool/OMNIbus WebGUI has the ability to transparently combine events from multiple IBM Netcool/OMNIbus systems into the same event views. The ability to deploy multiple IBM Netcool/OMNIbus systems and then combine the events at the IBM Netcool/OMNIbus WebGUI layer is one of the scalability features of the IBM Netcool suite. For more information on scaling your deployment by installing multiple IBM Netcool/OMNIbus systems, see the document entitled: *IBM Netcool/OMNIbus Large scale and geographically distributed architectures* referenced in the section *Your IBM Netcool/OMNIbus Best Practices library*.

Estimated number of users

IBM Netcool/OMNIbus WebGUI or IBM Netcool/OMNIbus Native Event List clients require periodic refreshes from the ObjectServer. These refreshes are implemented using SQL queries to the ObjectServer, and impose load on it. There is a limit therefore to the number of users an ObjectServer can support, hence the number of users an IBM Netcool/OMNIbus system is going to need to support is a significant sizing consideration.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Just as for any other connected client, IBM Netcool/OMNIbus WebGUI and Native Event List client load can be monitored via the ObjectServer's profiling log. The total time consumed by the user clients can vary dramatically depending on the complexity of the filters in use. IBM Netcool/OMNIbus WebGUI map pages, for instance, may have a great number of entities with complex filters behind them.

Note: Event List refreshes translate into read operations against ObjectServers and, although they incur load, read queries can and are overlapped by the ObjectServer. Hence including the profiling information for read clients does not provide an exact measure of ObjectServer load, rather an *indication* of load. The execution of SQL tools however does incur ObjectServer locks, however tool execution typically occurs less often than Event List refreshes, and so likely affects profiling values less.

As a general guideline, deploying Display layer ObjectServers should be considered if more than 10 users are to be supported, but only if there is noticeable lag when user displays are refreshed. Since IBM Netcool/OMNIbus 7.4, there has been a dramatic improvement in transactional concurrency in the ObjectServer for read operations. This means that it is unlikely that most deployments of modern versions of Netcool/OMNIbus will need an ObjectServer Display layer to resolve client-induced load performance issues.

Overall, the decision to put in Display layer ObjectServers should be taken based on profiling information and end-user experience. For example, if the hardware is very fast, the total event counts low and the user filters simple, an Aggregation ObjectServer may well support many more than 10 users without the need for a Display layer. Conversely, if the hardware is relatively slow, the total event counts high or the user filters complex, an Aggregation ObjectServer may support fewer than 10 users.

Note: IBM Netcool/OMNIbus WebGUI supports caching: where Active Event List (AEL) filter data sets are cached for a time-to-live period. Any other AEL clients logged in to the same IBM Netcool/OMNIbus WebGUI server using the same filter will automatically share the cached data instead of all clients individually fetching their own copy from the Display ObjectServer each time they refresh. This will further reduce the read load on the ObjectServer and therefore likely increase the number of supportable users. If users are looking at different event sets however, then the gains made by using caching will be reduced.

Just as for the other metrics, thorough testing is required with a representative number of users and the relative load checked before deploying any solution into production.

As a rough guideline, it is recommended to provision at least one Display ObjectServer per 100 users that will *concurrently* access the system. Display ObjectServers should be provisioned on an N + 1 basis to ensure resiliency. For example, if 30 concurrent users will connect to the system, 1 + 1 = 2 Display ObjectServers should be provisioned. If 130 concurrent users will connect to the system, 2 + 1 = 3 Display ObjectServers should initially be provisioned.

Provisioning guidelines summary

There are many factors that affect the provisioning of components in an IBM Netcool deployment. For example, the number and complexity of any custom triggers present may significantly affect the maximum number of events an IBM Netcool/OMNIbus infrastructure can support. Similarly, the variance and complexity of user filters can significantly affect the number of end users a resulting system will be able to support. What makes initial provisioning even more difficult is often business requirements are not fully finalised before the hardware needs to be ordered.

Nonetheless, it is necessary to be able to estimate an initial sizing based on the information that is known. This section contains suggested guidelines that can be used when performing initial provisioning calculations, so that an initial idea of how the deployment will look can be drawn up and initial hardware ordered.

Note that since the standard multitier architecture configuration is both highly scalable as well as modular by design, the architecture can be extended very easily if the induced load necessitates it.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

It is prudent when estimating initial provisioning therefore, to err on the side of provisioning a lower number of components, since additional components can be “bolted on” at a later time, if needed, due to the modular nature of the standard architecture configuration. Estimating a lower number of components and potentially having to add more components later is a typically a more favourable scenario than over-sizing a deployment and hence over-capitalising as a result.

The following table provides some general guidelines that can be used to produce a preliminary IBM Netcool/OMNIbus architecture sizing. Note that it does not consider any special considerations with respect to network layout, etc., it merely provides guidance on how to provision based on initial metrics gathered as part of the requirements gathering phase. Remember that these guidelines are provided to allow an *initial* sizing estimate to be done. It is likely that the initial sizing will have to be revisited, possibly several times, before the overall design is finalised.

Component	**Initial sizing guideline**	**Notes**
Probes	One Probe for every 100 events per second	• This guidance primarily applies to “listening” type Probes such as the SNMP Probe or the Syslogd Probe — that is, ones that passively receive events from multiple sources. • Temporary spikes of up to 500 events per second are acceptable as long as this rate is not sustained indefinitely. Probes are capable of buffering events should the incoming event rate exceed the rate at which it can process and send the events to the ObjectServer. • Note that an aggressive, automated event handling and purging mechanism should be in place to handle high event rates — otherwise the IBM Netcool/OMNIbus infrastructure will quickly be overwhelmed. See the section entitled: *ObjectServer housekeeping* later in *Chapter 4* for more guidance about how to go about implementing such schemes.
Aggregation ObjectServer pairs	One pair of Aggregation ObjectServers for up to 100,000 standing rows in the *alerts.status* table — 100,000 being the maximum recommended number of rows for a single IBM Netcool/OMNIbus partition	• Aggregation ObjectServers should be deployed in failover pairs: a primary and a backup connected by a bidirectional ObjectServer Gateway. • 100,000 rows should be considered an *absolute maximum* number of rows an Aggregation pair should hold. Ideally a system will have far fewer than this number since one must consider how many events network operations staff can practically monitor. This number can be used for provisioning purposes, however. • If the deployment will span more than one geographical region, consideration should be given to the geographically distributed architecture model. For more information on this type of deployment, see the document entitled: *IBM Netcool/IBM Netcool/OMNIbus Large scale and geographically distributed architectures - Best Practices*. More details of this document can be found in the section entitled: *Your IBM Netcool/OMNIbus Best Practices library* at the beginning of this document.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Collection ObjectServer pairs	One pair of Collection ObjectServers for each set of Probes that collectively send around 500 events per second	• Collection ObjectServers should be deployed in pairs: a primary and a backup — each with its own unidirectional ObjectServer Gateway connecting it to the Aggregation layer. There is no Gateway connection between the members of each failover pair. Each one does not have, nor does it need to have, any awareness of its counterpart. Each member of an ObjectServer Collection failover pair exists simply to provide two independent paths for events to be able to propagate from Probes, via the Collection layer to the Aggregation layer. • A Collection layer should be *considered* if the expected event rate is more than 100 events per second from all Probes. • Note that Collection layer ObjectServer provisioning is not based on the number of Probes — rather the collective event rate from the set of Probes. It is not connections that affect ObjectServer loading so much as the event rate.
Display ObjectServers	One Display ObjectServer for every 100 users on an N + 1 basis for resiliency	• Display ObjectServers should be deployed individually — each with its own unidirectional ObjectServer Gateway connecting it to the Aggregation layer. • A Display ObjectServer may be able to support more than 100 users if, for example: the user filters are efficient or the event numbers are relatively low — or a combination of factors such as these. • The converse is also true, however: a Display ObjectServer may be able to support less than 100 users if, for example: the user filters are inefficient or the event numbers are high — or a combination of factors such as these. • If Display ObjectServers are to be deployed, a minimum of two is required to ensure resiliency — hence N + 1.
JazzSM DASH servers running IBM Netcool/OMNIbus WebGUI	One "small" WebGUI server for every 50 users on an N + 1 basis for resiliency One "large" WebGUI server for every 90 users on an N + 1 basis for resiliency	• The same factors that affect the number of users a Display ObjectServer can support apply too to the number of users a WebGUI server can support. • Administratively standardising user filters and denying users the ability to create their own filters can significantly increase the number of users a system can support. • If operators typically use standardised filters, the WebGUI server can support substantially more users by enabling datasource caching. • If geographically remote datasources are included in WebGUI's configuration, or where the bandwidth to a remote datasource is limited, datasource caching is recommended for those datasources. Standardising the filter set used by operators, in this instance, is important to realise any benefit from caching.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: The difference between a "small" and "large" WebGUI server is around the amount of hardware resource allocated to it. See the section entitled: *CPU, memory, and disk space* resourcing for more details.

Where to implement custom functionality

In every deployment of IBM Netcool/OMNIbus, custom functionality is installed over top of the base configuration. Before planning to implement a piece of custom functionality however, consideration must be given as to what would be the most appropriate method of delivery. For example, would an ObjectServer trigger be the best place to implement *"requirement X"*? Or would it be more appropriate to implement the requirement within Probe rules files — or a IBM Netcool/Impact policy?

The following table provides some tips for how to decide where a piece of custom functionality should be implemented.

Implementation point	**Notes**
Probe rules file	The Probe rules file is where the incoming token stream is parsed and assigned to ObjectServer fields. It is a best practice to implement as much functionality as possible at the Probe rules file level — before it gets to the ObjectServer. This means the ObjectServer has less work to do when the event arrives. For example, dropping events that can be discarded is best done at the Probe rules file level. Note however that discarding events in at the Probe level is not always possible if, for example, a record of every event must be archived for legal reasons. In such scenarios, events can be sent to the Collection layer ObjectServers, forwarded to a historical archive, and then deleted from the Collection layer ObjectServer, without propagating up to the Aggregation layer. Alternatively, these log-only events can also be rerouted from the Probe directly to a separate, dedicated ObjectServer pair, so as not to affect the performance of the main Collection pair. The solution that is selected will depend on the event numbers and subsequent ObjectServer loading. For an example, see the section entitled *ObjectServer loading* on page 27 of this document. Lookup files are useful at the Probe level for basic event enrichment so long as the data set is static. If the data in the lookup table is dynamic, it is better to store it in a database table (eg. ObjectServer or other external database) and then perform event enrichment via an ObjectServer trigger or via IBM Netcool/Impact. Note however that creating large custom tables may cause ObjectServer performance to suffer. Custom tables and triggers, like any other custom functionality, should be thoroughly tested before being put into production.
ObjectServer database trigger	The database trigger is used for implementing actions based on changes to the event data set — for example: an insert of, reinsert of (deduplication), update to, or deletion of a row in an ObjectServer table. For example: a custom reinsert database trigger might be created on the *alerts.status* table to specify how a number of specified custom fields should be updated on deduplication. Care must be taken not to overload the actions of a database trigger as it may negatively impact on the ObjectServer's performance. Thorough testing and monitoring of database triggers in a test environment should be done to validate their performance however before deploying into production.
ObjectServer temporal triggers	The temporal trigger is used for implementing actions on a regular, timed basis. Common tasks of temporal triggers include: event correlation, event housekeeping, event flood detection and general event life cycle actions. Thorough testing and monitoring of temporal triggers in a test environment should be done to validate their performance before deploying into production.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServer signal triggers	System signals are "raised" spontaneously within the ObjectServer when certain system related scenarios occur — for example: when the ObjectServer starts up, shuts down, or when an external client connects. Signal triggers can be constructed to fire when a specific signal is raised to carry out predefined actions. Custom signals can also be created and can be a convenient way to organise triggers. Custom signals could be created for different scenarios — for example, *"max event count breached"*. Then, one or more signal triggers could be created and configured to fire when this signal is raised either manually — for example by an Administrator, or via an automated mechanism — for example: by a trigger or procedure.
External scripts	External scripts (launched by external action procedures from within the ObjectServer) are used for carrying out actions that require interaction with external systems. The `nco_mail` script is designed to be launched by an external action procedure to send a notification e-mail, for example.
IBM Netcool/Impact	IBM Netcool/Impact is an extremely powerful event processing engine and can be used for a great number of purposes including: • Event enrichment: typically where the target data resides on an external data source, such as a DB2 database; • Any sort of data correlation or data processing (event driven or on a timed basis) on data contained either in the ObjectServer or another source — for example: a file or a RDBMS; • Interaction with external databases; • Interaction with web services; • Interaction with external REST APIs; • Invocation of an automatic process or interaction with a system that requires timed delays (via the *Hibernation* function). Where custom functionality cannot be implemented by a Probe rules file or ObjectServer trigger due to the target system being external, consideration should be given to use IBM Netcool/Impact.

ObjectServer trigger or IBM Netcool/Impact policy?

It may be possible to implement a piece of custom functionality using either an ObjectServer trigger or an IBM Netcool/Impact policy. The question is, which is "best"? If all the information needed to carry out the required function is contained within the ObjectServer, first consideration should be given to implementing the function via an ObjectServer trigger.

First, although the ObjectServer is relatively very fast at carrying out tasks, it is important to note that an ObjectServer trigger runs as a single threaded process. Where performance is not at an acceptable level and all possible efficiencies have been made, it is worth considering implementing the function within Netcool/Impact, which is multi-threaded.

Second, if the load the trigger imposes on the ObjectServer starts to become significantly detrimental to the ObjectServer's performance, it may be prudent to move the processing load off the ObjectServer and into Netcool/Impact. Where the primary ObjectServer is a single entity, a cluster of Netcool/Impact servers is made up of multiple entities. If further processing "horsepower" is required within IBM Netcool/Impact, additional servers can be added to the cluster.

Third, there are occasions where the scope of the ObjectServer SQL language does not provide the functions or data structures necessary to implement a required function in an efficient manner. An example of an unsupported data type within the ObjectServer is a hash table. Although this data structure is available within an IBM Netcool/Impact policy, it is not available within the ObjectServer SQL language. Another example is the receipt and processing of JSON formatted payloads. New functions have been added to IBM Netcool/Impact since version 7.1 to enable it to easily parse and handle JSON payloads into a format that is more usable and consumable. JSON formatted payloads are common particularly when interacting with REST API targets.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Consider using scope-based event grouping

IBM Netcool/OMNIbus 8.1 comes with an automated event grouping feature called: "scope-based event grouping". This works by automatically grouping events together under a synthetic parent event than have: "come from the same place at the same time".

Event grouping can help to save a great deal of time for operations when resolving problems in your environment by automatically grouping events together that relate to the same problem. This helps an operator to determine the cause more quickly, if all the relevant problem and symptoms are organised logically together. It also enables just a single ticket to be cut for a group of events, saving a great deal of time and money. Scope-based event grouping also comes with a number of features and options that can be customised by modifying the relevant properties in the *master.properties* table.

The term "same place" is defined as whatever is entered into the *ScopeID* field in an event – for example: "LONDON". The *ScopeID* field is of type *VARCHAR* and so the *ScopeID* can be any non-null string.

Note: If the ScopeID field is not set for an event, no grouping is attempted.

The *ScopeID* can be set prior to an event's insertion into the ObjectServer – for example: in the Probe rules – or after insertion via an automation – for example: by an IBM Netcool/Impact policy. If the ScopeID field is set, the automation then automatically groups any events together that contain the same value in the *ScopeID* field, that have occurred at the same time.

The term "same time" is defined as a time window and can be either static or dynamic. The time variable used by scope-based event grouping is stored in an integer field called *QuietPeriod* and is used differently depending on the type of time window used.

The *QuietPeriod* can be defined dynamically in the incoming event stream or, if not defined (ie. 0), it will revert to the global value for quiet period, defined by the property *SEGQuietPeriod* in the *master.properties* table.

DYNAMIC TIME WINDOW

The dynamic time window is defined in terms of a "quiet period" during which no further events are received. If the system begins to and continues to receive a steady stream of new events for a given *ScopeID*, the time window and grouping remain open to new events. If the number of seconds defined by the *QuietPeriod* field passes however, without any further events, the grouping is closed, and no further events are added to the grouping. If any events from the given *ScopeID* are subsequently received after this time, a new grouping is automatically created, and the new events grouped under the newly created synthetic parent event instead.

When this mode of operation is used, the quiet period value defines the expiry time for the group. Whenever a new event is added to the group, the expiry time may be extended, if the current time plus the incoming *QuietPeriod* value is greater than the currently stored group expiry time. If current time plus the incoming *QuietPeriod* amounts to an earlier point in time, the automation will automatically retain the later timestamp for the group expiry time.

Note: Dynamic is the default grouping mechanism in scope-based event grouping.

FIXED TIME WINDOW

In some scenarios, a fixed time window is required. This is achieved simply by prepending the string "FX:" to the beginning of the *ScopeID* value – for example: "FX:LONDON".

When this mode of operation is used, the quiet period value defines the expiry time for the group, the same as for the dynamic time window. The difference is that when the group expiry time is set, it remains static, and is not dynamically updated as new events enter the group. When the group expiry time passes, the grouping is closed, and no further events are added to the grouping. If any events from the given *ScopeID* are subsequently received after this time, a new grouping is automatically created, and the new events grouped under the newly created synthetic parent event instead.

--

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Regardless of which time window method is used, scope-based event grouping can be used simply by setting *ScopeID* and, optionally, *QuietPeriod*. It is therefore a very convenient, high-value out-of-the-box mechanism that can be leveraged with very little configuration.

Scope-based grouping also includes a number of other features such as: automatic propagation of ticket number (*TTNumber*), *Acknowledged* status, *OwnerUID*, and *OwnerGID* to child events; automatic journaling of child event details to the synthetic parent journal.

Note: For a full listing of properties and installation instructions, please refer to the IBM Netcool/OMNIbus 8.1 documentation: http://ibm.biz/seg_docs

Architectural considerations

When attempting to calculate the number of Netcool components required during the design of a IBM Netcool architecture, the previous sections referred to variables such as: number of users, number of standing events, number of events per day, and so on.

In addition to these factors, consideration should also be given to the geographical layout of the locations to be managed, security and network restrictions, resiliency, and the loading incurred by custom functionality. Additional components may also need to be added to resolve architectural or business rule constraints.

Geographical layout

The *physical location* of the components to be managed will usually affect the geographical layout of the resulting Netcool architecture, as well as the *number* of Netcool components required to implement the design. Additional components, such as Collection ObjectServers, are not always *only* added for load reasons, as the following examples illustrate.

EXAMPLE:

Widgetcom's remote site “A” has two peered SNMP Probes receiving 5,000,000 traps per day. The devices sending these traps are very chatty, and many of the events deduplicate. The Netcool Administrator has two concerns: first, that traps transmitted via UDP across the WAN to the Aggregation ObjectServers located in site “B” may be lost due to the nature of UDP being non-guaranteed delivery; second, the relatively high amount of data these Probes are sending over the WAN needs to be reduced. The Netcool Administrator decides to install a primary/backup pair of Collection layer ObjectServers at site “A” for the Probes to connect to locally. Not only will the Collection ObjectServers deduplicate the events before transmission over the WAN, the Collection to Aggregation Gateways will use TCP instead — ensuring reliable delivery.

EXAMPLE:

Widgetcom has two sites — one in London (United Kingdom) and one in Wellington (New Zealand). The requirement is to have visibility of all events from both sites — when the network allows. The Netcool Administrator decides to deploy two separate Netcool instances: an Aggregation failover pair of ObjectServers at the London site and an Aggregation failover pair of ObjectServers at the Wellington site. In addition, each site has two (for resiliency) IBM Netcool/OMNIbus WebGUI servers. The IBM Netcool/OMNIbus WebGUI servers are configured with two datasources: `LONDON` and `WELLINGTON`. The main Active Event List view on all IBM Netcool/OMNIbus WebGUI servers is then configured to include events from both datasources. Finally, caching is enabled for the remote datasource in each case to minimise the data flowing across the WAN.

Security and network restrictions

Very often, security and network restrictions or limitations will also affect an IBM Netcool architecture design and numbers of components.

EXAMPLE:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Widgetcom have a number of Probes deployed in the DMZ network and their Aggregation ObjectServers deployed in their core network. These two networks are separated by two firewalls with an intermediary network in between. Network security policy stipulates that the Probes located in the DMZ network are not allowed to establish connections directly through both firewalls to the Aggregation ObjectServer pair located in the core network. The Netcool Administrator decides to deploy a pair of IBM Netcool/OMNIbus Firewall Bridges in the intermediary network which will then connect to the Aggregation layer ObjectServers. The Probes in the DMZ can then connect to the Firewall Bridges instead.

EXAMPLE:

Widgetcom have a large number of monitored devices at remote site "A" and Probes to monitor those devices. The Aggregation ObjectServers are meanwhile located at the main site: site "B". The Probes are very chatty, however most events deduplicate to a very small number. The network bandwidth between the sites A and B is relatively small, and there is concern that the network pipe over the WAN will not support the event rate from the monitored devices. The Netcool Administrator therefore decides to deploy a Collection pair of ObjectServers at site A. The events coming from the Probes will then deduplicate within the Collection ObjectServer pair and forward only a deduplicated summary of the received event set across the WAN, rather than each individual instance of each event.

Resiliency

High availability and resiliency is hallmark of Netcool product suite, and therefore is an important consideration for any Netcool deployment. It is a best practice that every Netcool component be deployed in a *resilient* manner. With respect to ObjectServers, it means deploying ObjectServers in pairs (or in groups of two or more for Display layer ObjectServers); for Probes, it means deploying them in pairs and then peering them as master/slave pairs (where appropriate); for Gateways (other than standard multitier architecture configuration), it means deploying them with a cold stand-by in place.

EXAMPLE:

Widgetcom have a number of SNMP enabled devices that they want to configure to send traps to the SNMP Probe. To ensure resiliency, the Netcool Administrator has installed a second SNMP Probe on a different host machine and has configured master/slave peering between the two. The managed entities have been enabled to send traps to both Probes. The slave Probe will automatically discard any traps received while the primary Probe is active.

EXAMPLE:

Widgetcom have an IBM Netcool/OMNIbus installation that supports 40 operators at any one time. Recently however, they scaled up their operations and now need to support 150 users. Since doing so, the operators reported Event List refresh times to be "sluggish". ObjectServer profile logs confirm that the Aggregation ObjectServer is spending a great deal of time servicing Event List requests. The Netcool Administrator decides to deploy a pair of Display layer ObjectServers to support the user load. Although a single Display ObjectServer would be able to handle the load on its own, it would become a single point of failure, hence a pair of Display ObjectServers is necessary. The Display ObjectServer pair is installed on two separate host machines, in two separate buildings, for resiliency.

Note: To ensure the maximum level of resiliency and disaster recovery ability, primary and backup components should be installed on separate hardware and, where possible, on different sites from each other.

Note that in some cases, it does not always make sense to have back up components:

- Collection to Aggregation ObjectServer Gateways are normally physically located on the same machine as the Collection ObjectServer. If the host machine were to go down therefore, a backup Collection to Aggregation Gateway for that Collection ObjectServer would be redundant since the Collection ObjectServer would be down also.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- A Syslog Probe running on a host machine connects directly to that host's system log. If the host were to go down, the system log would be unavailable. Any sort of backup Syslog Probe therefore in this scenario would therefore be redundant.

Common sense should be applied to all architectural design decisions.

ObjectServer loading due to custom functionality

Once IBM Netcool/OMNIbus has been deployed using the standard multitier architecture model, custom functionality is normally applied over the top. The loading incurred by custom functionality may be significant and, in some cases, it may be desirable to deploy additional ObjectServers to help support the additional load. Only thorough stress testing at the end of the development phase will validate if a design is fit to support the loading metrics. Where deficiencies are found, additional components can be added to support the load.

EXAMPLE:

Widgetcom receive a large number of events per day and this is causing a high load on the Aggregation ObjectServers. Although they have a legal obligation to store all the events they receive for 3 months, the operators only need to see and act on around 10% of them. They use an IBM Netcool JDBC Gateway to forward all received events to a DB2 database. Any events that do not need to be seen by the operators are deleted after they have been archived. The Netcool Administrator decides to deploy Collection layer ObjectServers each with its own Netcool JDBC Gateway connected. All events are received by the Collection ObjectServers initially and are all sent to DB2 for archiving. After this step, only events that need to be seen or acted on by operators are sent up to the Aggregation layer — the rest are deleted. This results in a *significantly* reduced loading on the Aggregation ObjectServers.

EXAMPLE:

Widgetcom receive a large number of events per day and this is causing a high load on the Aggregation ObjectServers. On inspecting the ObjectServer profiling information, IBM Netcool/Impact is shown to be consuming a great deal of the ObjectServer's time. The IBM Netcool/Impact servers are performing both one-time event enrichment and event correlation. The Netcool Administrator decides to configure IBM Netcool/Impact to carry out the one-time event enrichment at the Collection layer instead, thereby lifting some of the processing load off the Aggregation layer. After making this change, the profiling on the Aggregation layer is reduced, allowing the Aggregation ObjectServers more cycles for other tasks.

Note: Correlation type functionality can only effectively be implemented at the Aggregation layer, since that is the only place where a trigger or IBM Netcool/Impact policy will have visibility of all current events. Since events are reaped from the Collection layer once they have been passed to the Aggregation layer, there is not the same visibility of events at the Collection layer as there is at the Aggregation layer. Typically, only one-time enrichment type functionality can be implemented at the Collection layer therefore.

Hardware considerations

Two main areas of planning an IBM Netcool/OMNIbus deployment with respect to hardware are resiliency and resourcing.

Hardware resiliency

Each part of an IBM Netcool/OMNIbus solution should be resilient, both in terms of software *and* hardware. Software resiliency is implemented by having backup components available to take over should the primary component be unavailable — for example: a failover pair of ObjectServers. Hardware resiliency is implemented by not having both primary and backup components for any one service (eg. Aggregation layer ObjectServers) either collocated on the same physical server, or in the same physical location.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

When including hardware resiliency into an architectural design, the following points should be considered so as to avoid having any potential single points of failure:

- Loss of a physical or virtual server a component is running on;
- Loss of an entire site where one or more components may be located.

EXAMPLE:

Widgetcom are planning an IBM Netcool/OMNIbus deployment; initially with just a single Aggregation pair of ObjectServers. They have selected two sites located in different parts of the city as part of their Disaster Recovery plan. The Netcool Administrator decides to install a server on each site and run the primary Aggregation ObjectServer on one server on one site — and the backup Aggregation ObjectServer and bidirectional ObjectServer Gateway on the server on the other site. This provides both functional and physical resiliency.

CPU, memory, and disk space resourcing

Once you have identified the number of servers you will need and their respective physical placement to ensure resiliency, you will need to calculate the number of cores and memory each server will need. The table included in this section gives recommendations for the number of cores and memory provisioning requirements for each of the components.

In order to calculate a server's hardware requirements to run one or more of the above components, simply add up the total number of cores and gigabytes of RAM, then add allowance for any other processes that might also be running on the target machine.

From IBM Netcool/OMNIbus version 7.4 onwards came the first 64-bit versions of the product. IBM Netcool/OMNIbus 8.1 is only available as a 64-bit version. These 64-bit versions allow for far greater overall process size than those listed above however one must be mindful of how table row count can affect ObjectServer performance. Simply because the process can address seemingly limitless amounts of memory, doesn't mean the ObjectServer can suddenly accommodate more events than before.

Note: The 64-bit version of IBM Netcool/OMNIbus will not provide any performance gains over the older 32-bit versions.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Component	Recommendation	Notes
Collection layer ObjectServer	Cores: 2 RAM: 4 GB	
Aggregation or Display ObjectServer	Cores: 4 RAM: 4 GB	
ObjectServer Gateway	Cores: 1 RAM: 2 GB	ObjectServer Gateways are highly efficient at transferring events but are bound by the same CPU and memory limitations as the ObjectServer. They are not subject to the same connection expectations as the ObjectServer however.
Third-party Gateways	Cores: 2 RAM: 4 GB	ObjectServer to third-party database or application Gateways can be more CPU and memory intensive than ObjectServer Gateways due to the connection mechanisms of the target database or application. Examples include the JDBC Gateway, the Remedy Gateway, or the Oracle Gateway.
“Listening” Probes (eg. SNMP, Socket, ...)	Cores: 2 RAM: 2 GB	Probes that “listen” on a port are lightweight but are typically multi-threaded (ie. able to accept and process incoming events on separate threads) so should be allocated 2 cores and plenty of RAM to handle event storm conditions. If the probe is using a complex rule set (such as the Netcool Knowledge Library — NcKL) then a further 0.5GB RAM should be allocated.
“Logfile” or “Target” Probes (eg. Syslog, Corba, SCOM, ...)	Cores: 1 RAM: 2 GB	Probes that connect to a “target” or read from a “logfile” are typically less CPU intensive as “listening” Probes due to the fact that events are processed in a single stream. The same Memory recommendations apply however.
“Small” IBM Netcool/OMNIbus WebGUI server	Cores: 2 RAM: 4 GB	A “small” WebGUI server can support up to 50 users.
“Large” IBM Netcool/OMNIbus WebGUI server	Cores: 4 RAM: 4	A “large” WebGUI server can support up to 90 users.

Note: See the *IBM Netcool/OMNIbus sizing guide* on the IBM Netcool/OMNIbus wiki for more information about the sizing information provided in this section:

https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Tivoli%20Netcool%20OMNIbus/page/OMNIbus%20Sizing%20Guide

EXAMPLE:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Widgetcom needs to deploy a backup Aggregation ObjectServer host machine. The software components would be one Aggregation ObjectServer and one bidirectional ObjectServer Gateway. The initial hardware provisioning would therefore be 4 + 1 = 5 cores and 4 + 2 = 6 GB RAM.

——

It is recommended that in hardware selection, Intel® based processors be 3 GHz or higher and RISC based processors be 1.5 GHz or higher. Previously the ObjectServer and other IBM Netcool/OMNIbus components have not been well suited to servers with multiple low speed CPU cores. Although improvements have been made with IBM Netcool/OMNIbus 8.1 on these types of servers, ObjectServers should ideally be run on a host with as high speed CPUs as possible, so that ObjectServer write operations can be performed as quickly as possible. This will be constrained by budget, of course.

The ObjectServer is a multi-threaded application, however its ability to utilise multiple processors is highly dependent on the tasks it is carrying out. Previously a version 7.x ObjectServer could utilise multiple processors while carrying out multiple, concurrent read operations however a write operation would require an exclusive lock to the entire data store locking out all other threads, including read operations, for the duration of the write. An example of this might be a trigger firing; or an event being inserted by a Probe.

From IBM Netcool/OMNIbus version 7.3.1 onwards however, there is a more relaxed locking model. The ObjectServer will still only allow a single write thread to execute at any one time, however it performs a finer grain locking, rather than locking the entire data store. A read operation will only be made to wait if it is trying to read a row that is being written to by a write operation at that same moment. Not only does this greatly reduce the incidence of thread clash, any waits incurred will be for a much-reduced moment of time when it does happen.

Because of the improved concurrency model in version 7.3.1 onwards, the ObjectServer is more capable of utilising more CPU cores. This enhancement allows IBM Netcool/OMNIbus to run more effectively on multi-cored server architectures, whilst also allowing administrators to scale the application more simply by adding additional processor core resource to existing servers.

In normal operating conditions, a standalone ObjectServer will typically use 1 to 2 CPU cores. The benefits of additional CPU cores are most evident for ObjectServers with a lot of concurrent connections (Aggregation or Display for example) and particularly in flood or failover conditions where the ObjectServer is worked hard.

Regarding disk space requirements, an IBM Netcool/OMNIbus installation on a host should be given approximately 20 GB in addition to the install footprint to allow for log files and ObjectServer checkpoint files.

Note: The installation of IBM Netcool/OMNIbus requires between 1.1 and 1.4 GB of disk space depending on the installation platform. Please refer to *Chapter 4: Planning for installation or upgrade* of the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide* for more specific information.

Example hardware specifications

This section contains some suggestions for hardware specifications when using the standard multitier architecture configuration. It uses the sizing information provided in the previous section to enable the calculation of how each machine type should be provisioned; given the components that would typically be installed on it.

Machine Type	**Components**	**CPUs/cores**	**Memory**	**Disk**
Probe server (master or slave)	2 × “listening” Probes 4 × “target” Probes	2 × 2 CPU/core + 4 × 1 CPU/core **= 8 CPUs/cores**	2 × 2 GB + 4 × 2 GB **= 12 GB**	**50 GB**

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Collection ObjectServer (primary or backup)	1 × ObjectServer 1 × unidirectional ObjectServer Gateway	1 × 2 CPUs/cores + 1 × 1 CPU/core **= 3 CPUs/cores**	1 × 4 GB + 1 × 2 GB **= 6 GB**	**50GB**
Aggregation ObjectServer (primary)	1× ObjectServer	1 × 4 CPUs/cores **= 4 CPUs/cores**	1 × 4 GB **= 4 GB**	**50 GB**
Aggregation ObjectServer (backup)	1× ObjectServer 1 × bidirectional ObjectServer Gateway	1 × 4 CPUs/cores + 1 × 1 CPU/core **= 5 CPUs/cores**	1 × 4 GB + 1 × 2 GB **= 6 GB**	**50 GB**
Display ObjectServer	1 × ObjectServer 1 × unidirectional ObjectServer Gateway	1 × 4 CPUs/cores + 1 × 1 CPU/core **= 5 CPUs/cores**	1 × 4 GB + 1 × 2 GB **= 6 GB**	**50GB**

Note: In a virtualised environment, it is possible to assign odd numbers of CPUs/cores to a virtual machine — for example: 3 or 5 CPUs/cores. This is not necessarily practicable to do this when dealing with physical standalone machines however. In the case of the latter therefore, it is recommended to simply round the calculated number in the above examples up. For example: where 3 CPUs/cores are recommended, a 4 CPU/core machine might be provisioned instead.

Setting up file descriptors

UNIX based platforms usually limit the number of file descriptors available to processes. For Netcool/OMNIbus, it is recommended to ensure the minimum number allocated is 1024.

EXAMPLE

The Netcool administrator is preparing a Linux server to run Netcool/OMNIbus. She checks the current process file descriptor process as follows:

```
[netcool@omnihost~]$ ulimit -n
1024
[netcool@omnihost ~]$
```

In this case, the soft limit is set to 1024, which is suitable for running IBM Netcool/OMNIbus.

Note: The process for modifying the file descriptors available to processes varies from platform to platform. See the relevant operating system documentation for more information on making any necessary changes therefore, of required.

Network resourcing

With respect to network requirements, components should be in a data centre with good network reliability and bandwidth. Network connection speeds of 100 Megabits per second or higher are typically sufficient.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Historical event archive sizing guidance

It is a common requirement to maintain a historical event archive that reports can be run against. Indeed, the Seasonality and Related Event functions in the Netcool Operations Insight offering has the REPORTER historical event archive as a prerequisite component.

When planning an IBM Netcool/OMNIbus deployment therefore, it is likely that a historical event archive will need to be sized and deployed.

An SQL file is provided with the Netcool database Gateways to construct the historical event archive schema in each case. What a database administrator will need to know is an idea of the space required to store the event data.

The calculation starts with a "typical" event size (in bytes), a typical journal size and a typical detail size. These values are then multiplied by the number of expected events per day and then multiplied by the number of days' worth of data that needs to be retained.

The Netcool/OMNIbus Gateway for JDBC Guide contains guidance on how to calculate the space required in a database to house the historical event data.

Note: The Netcool/OMNIbus Gateway for JDBC Guide can be accessed here: http://www.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/gateways/jdbcgw/wip/concept/jdbcgw_intro.html

---

REPORTER MODE

For a Gateway running in Reporter mode (which is the most common), the following formula is suggested. Note that the audit tables are recording only one entry per event as a baseline.

```
<inserts per day> * (<bytes per event> +
(<number of audit tables> * <bytes per audit table row>) *
<52 weeks> * <7 days>) / <bytes per GB>
```

For example, using the following values, the required space for one year's worth of events would be:

- 10,000 inserts per day, after deduplication
- 2048 bytes per event
- 4 audit tables
- 256 bytes per audit table row
- 52 weeks
- 7 days

This equates to:

(10,000 * (2,048 + (4 * 256)) * 52 * 7) / 1,024^3 = 10.4 GB

If there will be on average two journals and two details per event at 512 bytes each, this equates to an additional:

(10,000 * 4 * 512 * 52 * 7) / 1,024^3 = 6.9 GB

Add these together to get: 17.3 GB.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: The values given are typical in customer deployments. You may need to modify these if they vary significantly in your environment – for example: journals may be more widely used than the assumptions given.

It is recommended to add an amount of contingency to the resulting figure to cater for higher than expected event numbers. In this case, by adding a contingency factor of 20%, we arrive at a total of: 20.8 GB.

AUDIT MODE

For a Gateway running in Audit mode, the formula slightly different. First, there are no audit tables and second, a row is created in the historical archive for each insert and reinsert:

```
(<inserts and reinserts per day> * <bytes per event> *
<52 weeks> * <7 days>) / <bytes per GB>
```

For example, using the following values, the required space for one year's worth of events would be:

- 10,000 inserts per day
- 40,000 reinserts per day
- 2,048 bytes per event
- 52 weeks
- 7 days

This equates to:

((10,000 + 40,000) * 2,048 * 52 * 7) / 1,024^3 = 37.3 GB

If there will be on average two journals and two details per event at 512 bytes each, this equates to an additional:

(10,000 * 4 * 512 * 52 * 7) / 1024^3 = 6.9 GB

Add these together to get: 44.2 GB.

It is recommended to add an amount of contingency to the resulting figure to cater for higher than expected event numbers. In this case, adding a contingency factor of 20% results in a total of: 53.0 GB.

Housekeeping historical event archive data

Periodically a database administrator will need to purge old historical event data from the historical archive database. The following sample SQL shows how to delete data from the historical event archive that is older than 60 days for both DB2 and Oracle.

The three key tables are the reporter_status, reporter_journal and reporter_details tables. Where the Gateway is running in REPORTER mode, the audit tables that track changes in Severity, Acknowledged state, who the OwnerUID and OwnerGID also need to be cleaned. Where the Gateway is running in AUDIT mode however, this is not necessary.

DB2

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
delete from reporter_status where
                     lastoccurrence < CURRENT DATE - 60 DAYS;
delete from reporter_details where identifier not in
                           (select identifier from reporter_status);
delete from reporter_journal where
                     chrono < CURRENT DATE - 60 DAYS;
delete from rep_audit_severity where
                     enddate < CURRENT DATE - 60 DAYS;
delete from rep_audit_ack where
                     enddate < CURRENT DATE - 60 DAYS;
delete from rep_audit_ownergid where
                     statechange < CURRENT DATE - 60 DAYS;
delete from rep_audit_owneruid where
                     statechange < CURRENT DATE - 60 DAYS;
```

ORACLE

```
delete from reporter_status where
                     trunc(lastoccurrence) < trunc(sysdate) - 60;
delete from reporter_details where identifier not in
                           (select identifier from reporter_status);
delete from reporter_journal where
                     trunc(chrono) < trunc(sysdate) - 60;
delete from rep_audit_severity where
                     trunc(enddate) < trunc(sysdate) - 60;
delete from rep_audit_ack where
                     trunc(enddate) < trunc(sysdate) - 60;
delete from rep_audit_ownergid where
                     trunc(statechange) < trunc(sysdate) - 60;
delete from rep_audit_owneruid where
                     trunc(statechange) < trunc(sysdate) - 60;
```

Other installation prerequisites

- It is recommended to always run with the latest fix pack available. Before installing IBM Netcool/OMNIbus, check on the IBM Netcool/OMNIbus Technical Support site and download the latest fix pack available. Alternatively, you can monitor the IBM Netcool/OMNIbus Technical Support RSS feed for notifications of all bugs, technical tips, and fix pack/interim fix availability information. Each RSS feed includes download locations for any items published.

  The IBM Netcool/OMNIbus Technical Support RSS feeds can be accessed here:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

http://www.ibm.com/software/support/rss/tivoli/

- All IBM Netcool/OMNIbus ObjectServers, Probes and Gateways should be configured to run under Netcool Process Agent control. See the section entitled *The Netcool Process Agent and machine start-up* later in this document for more detailed information regarding the Netcool Process Agent.
- No Netcool components should be left running in debug mode in a production environment unless under the direction of IBM Technical Support.
- Automated housekeeping functionality should be engineered into the solution to ensure that events cannot remain indefinitely in the ObjectServer. Establishing an event expiry policy for all types of events is a good way of achieving this. Expiry times can be set in the events at the Probe level — with a "catch-all" trigger in the ObjectServer to set the `ExpireTime` field for events where it is not set. The default trigger `expire` will then remove events once their expiration time has passed. See the section entitled *ObjectServer housekeeping* later in this document for more detailed information regarding ObjectServer data housekeeping.

Solution delivery

Consideration as to how an IBM Netcool solution will be delivered by a solution engineering team should be given during the planning phase of any deployment project. The delivered solution should include custom configuration files and all relevant documentation. The concepts introduced in this section will be referred to throughout this document.

All deliverables should be reviewed with the end-customer before final project acceptance and sign-off. The "end-customer" will normally be the primary stakeholder and owner of the resulting IBM Netcool deployment.

Custom configuration

The following list summarises the best practice approach for delivery of each of the custom configuration components in a solution, and ensures that a solution is as portable and modular as possible.

All the items mentioned below should be collected together into a single directory location and packaged up together (ie. using a `zip` or `tar` utility) for delivery along with the documentation described in the next section.

It is recommended that each of the following IBM Netcool/OMNIbus components be delivered as follows:

- All *Native Event List* custom menus, prompts and tools should be developed on the development system and then exported to a package using the `nco_confpack` utility. This can be found in the `$OMNIHOME/bin/` directory. Note that only custom components should be included in the solution delivery package. Default menus, prompts and tools are created at ObjectServer initialisation time (ie. `nco_dbinit`) and should not be included in any solution delivery packages, only the custom ones.
- All other custom ObjectServer internal components including custom fields, custom log file definitions, custom signals, custom procedures, and custom triggers should be written up in ObjectServer SQL into a plain text file as per the examples given throughout this document, and given a `.sql` file extension. Note that, as in the case of the standard multitier architecture configuration, a separate solution delivery file will usually be required for each tier when multiple tiers are deployed.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This file or group of files is called the *Solution Delivery File(s)* and is a term referred to extensively within this document. See the multitier configuration ObjectServer SQL files for examples of solution delivery files found in the following directory:

```
$OMNIHOME/extensions/multitier/objectserver/
```

When building solution delivery SQL files, take care that the various custom components are created in an order that respects *dependencies*. For example, custom fields that are referenced in triggers should be created before the triggers themselves. Hence it is recommended that *all* custom fields are created in the custom delivery SQL file before any of the triggers. For the sake of readability and maintainability, items in the solution delivery files should be logically grouped where possible.

Note that solution delivery SQL files are used by the product to create vanilla ObjectServers when the `nco_dbinit` command is called. The SQL files used by `nco_dbinit` can be found in the following directory

```
$OMNIHOME/etc/*.sql
```

The default solution delivery SQL files contained in `$OMNIHOME/etc/` should not be modified or updated in any way.

- All Gateway configuration files used in the solution should be captured and included in the solution delivery package.
- All Probe properties, rules, include, and lookup files used in the solution should be captured and included in the solution delivery package.
- All Native Event List view (`.elv`), filter (`.elf`) and configuration (`.elc`) files should be captured and included in the solution delivery package.
- If the Netcool/OMNIbus Initial Configuration Wizard (ICW) has been used, the solution deployment descriptor file should be included in the solution delivery package.

NOTES:

- Solution portability is important when deploying the solution to multiple locations — for example, when replicating an environment from development, to test and then to production. A portable solution also makes it very easy for the environment to be recreated elsewhere — for example: by IBM Netcool Technical Support engineers in their labs. Following the guidance above will result in a very portable solution.
- All custom functionality should be kept *separate* from default functionality wherever possible. This is important so that if any changes are made to default componentry — for example: in a fix pack, or a new IBM Netcool/OMNIbus release — then those updates to default components will not inadvertently overwrite any customisations when applied.

  For example, if additional deduplication behaviour is required, the default deduplication trigger (ie. `agg_deduplication`) should not be disabled or modified. A new deduplication trigger should be created instead. An ObjectServer can happily have more than one reinsert trigger present. Where two triggers are configured to fire at the same time — for example: two reinsert database triggers acting on the same table, the order that they fire will be determined by the priority.
- In addition to solution delivery SQL files, rollback files should also be produced and included in the final delivery package. A rollback file will do as the name suggests and undo the modifications the solution delivery SQL file makes to an ObjectServer. Care must be taken to remove the custom components in the reverse order that they were created. For example, before

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

dropping a custom field, ensure that all triggers and other components that reference the field are removed first. For examples of rollback SQL files, see the standard multitier architecture configuration files located in:

```
$OMNIHOME/extensions/multitier/objectserver/
```

- One method for exporting configuration from an existing ObjectServer as SQL is to use the `nco_osreport` utility with the `-dbinit` switch. Use the `-help` switch to see other available options for its use. The resulting SQL file output should be reviewed, the relevant sections captured and commented appropriately (see the section entitled *Custom triggers* later in this document) and then included in the solution delivery SQL file.

Documentation

Clear and thorough documentation is a key to the success of any solution delivery. The solution delivery documentation would normally be produced by the engineering team responsible for delivering the solution. The documentation set is essential for anyone reviewing, deploying, or maintaining the solution.

The Business Requirements Document

The first key document is the *Business Requirements Document* (BRD). The BRD should contain a detailed description about every piece of custom functionality that will be required by the Netcool deployment, including pseudocode, where possible.

A prerequisite to the production of the BRD would be for the solution delivery team to undertake discussions with the business management and stakeholders in order to compile a detailed list of the business requirements. The BRD is essential for ensuring that solution scope does not creep during the solution development phase.

An example requirement pseudocode entry follows:

Critical events must be escalated if unacknowledged after 10 minutes:

- *Check all critical events every 2 minutes*
- *For each one that is more than 10 minutes old but has not yet been acknowledged*
- *Send an e-mail alert to the operations manager including the event details*

Note that the BRD may include requirements that will need more than just IBM Netcool/OMNIbus to provide. For example, a requirement may need both IBM Netcool/OMNIbus and IBM Netcool/Impact to implement. The implementation details are not included in the BRD however, just the requirements themselves.

The Detailed Design Document

The second key document is the *Detailed Design Document* (DDD). The DDD should contain a detailed, low-level implementation description for each business requirement in the BRD, specifying how each requirement will be implemented and the rationale behind each implementation method, since there are usually many ways to implement each requirement.

The DDD will also usually include the actual code that will be found in the solution delivery SQL files. Note that the DDD may well include implementation details of each requirement across multiple products, and not just IBM Netcool/OMNIbus.

The DDD is used as a baseline reference that the custom solution would be engineered against, both by the solution delivery team and engineers that work on the system after initial deployment.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Solution Delivery Package Document

The third key document is the *Solution Delivery Package Document* (SDPD). The SDPD should contain a detailed, itemised listing of each of the components in the solution delivery zip or tar archive, and detailed instructions of how to install or apply each one. Note that the solution delivery package may contain more than just IBM Netcool/OMNIbus components — therefore instructions should be provided in the SDPD for the installation/application of all the parts.

The SDPD is a reference for the un-packaging and deployment of each of the custom functionality components and is intended for use by anyone building out the delivered system either in a development, test, or production environment.

Note: If the solution delivery package contains implementation components for more than one product, subdirectories should be created within the package for each product (eg: `omnibus/`, `impact/`, etc.)

The documentation should be produced in the order listed above — that is: the BRD first, then the DDD and finally the SDPD. If there are any necessary changes to the requirements at any stage of the development phase, the documents will have to be reviewed and modified in the same order as they were initially produced.

Use of debug mode

The insertion of debugging code and the running of components in debug mode are both useful tools in Netcool solution development.

Debugging code may come in the form of additional lines of code in ObjectServer triggers or Probe rules files that write additional information to log files, for example. It is not recommended to include code such as this in a production environment as it may invariably cause a performance bottleneck. File I/O operations (ie. writing to log files), for example, can be relatively costly.

No IBM Netcool/OMNIbus component should be left running in debug mode in a production system. Debug mode significantly degrades performance.

On extremely rare occasions, it is sometimes necessary to run an IBM Netcool process in debug mode in a production system when attempting to triage a critical issue — however once a diagnosis is made and the problem resolved, those components must be taken out of debug mode and run normally.

Stress testing prior to deployment into production

It is essential to perform stress load testing on an IBM Netcool/OMNIbus solution prior to deployment into production to ensure that it will cope in an event storm scenario. Stress load testing involves loading up the system with a maximum peak amount of data, including maximum peak number of events (in `alerts.status`), journals (`alerts.journal`), details (`alerts.details`) and maximum expected quantities of data in custom tables all at the same time — and then monitoring ObjectServer profiling, user interface responsiveness and the component log files for any potential performance related problems.

Note: When loading up the IBM Netcool/OMNIbus system with data, it is important to use a *representative* set of data similar to what production would likely hold. This will ensure that memory usage and trigger load will be similar to what can be expected. There is limited value in stress testing a system with synthetic events that do not match any triggers.

Checking the table_store memstore hard and soft limits

One item in particular to check on during stress load testing is the usage of the table memory store when the system is loaded up to its peak maximum, particularly in the Aggregation layer

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServers. This can be done by checking the `catalog.memstores` table as per the following example:

```
1> select UsedBytes, SoftLimit, HardLimit from catalog.memstores
where StoreName = 'table_store';
2> go
 UsedBytes   SoftLimit   HardLimit
 ----------- ----------- -----------
     1348112   471859200   524288000
```

The information in the above example shows that the amount of used space within the ObjectServer table store is 1,348,112 bytes while the maximum soft limit is 471,859200 bytes and the maximum hard limit is 524,288,000 bytes. In this case, the ObjectServer is using only a fraction of its maximum available space.

Note: The above query can be made via the `nco_sql` prompt or via the SQL tab in the Netcool Administrator. You can also browse the contents of the `catalog.memstores` table via the Databases tab of the Netcool Administrator to see these values.

The soft and hard limits in this example are the default IBM Netcool/OMNIbus values and reflect the values set by following line in the default `$OMNIHOME/etc/system.sql` file that is used when the ObjectServer is initialised via the `nco_dbinit` utility:

```
create memstore table_store persistent hard limit 500M soft limit
450M;
```

When the soft limit is breached, you can expect to see a message like the following in the ObjectServer's log file:

```
Region soft limit exceeded
```

If this occurs, you can use the ALTER MEMSTORE command to change the soft limit via an SQL command prompt or from the SQL tab in the Netcool Administrator GUI:

```
ALTER MEMSTORE table_store SET SOFT LIMIT 500 M;
```

Note: You cannot set the soft limit to a value bigger than the hard limit.

If you need to, you can increase the size of the ObjectServer’s memory store hard limit via the `nco_store_resize` utility. For most environments, this is not usually necessary as the default values are sufficient. Normally ObjectServer performance will suffer long before the table stores are full.

The maximum size for ObjectServer hard limits depends on the platform. Before making any changes to an ObjectServer's hard limit via the `nco_store_resize` utility, always back up the database files in the `$OMNIHOME/db` directory.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The maximum permitted store size (in MB) by operating system is as follows:

- AIX: 2047
- Solaris: 2047
- Linux x86: 2047
- HPUX: 1024
- zLinux: 1024
- Windows: 700

If you try to increase the hard limit to a value greater than the above values, you may find your ObjectServer will not start. The `store_resize` process is irreversible also, which is why it is *imperative* to take a backup of your ObjectServer database files before using it.

Note: See the section entitled *Changing the table_store memstore soft and hard limits* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more information on how to change the ObjectServer's table store memstore.

Removal of temporary files

A final note, after deployment of a delivered solution into an environment, any temporary files should be removed to leave a clean directory structure. Leaving temporary files sitting around in Netcool directories is unprofessional.

Note that the term *"temporary files"* does not include backup files — for example: when iterative changes are made to a Probe's rule file and a backup taken.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 3 Installing and upgrading

Full instructions for installing IBM Netcool/OMNIbus can be found in *Chapter 5: Installing and updating IBM Netcool/OMNIbus* in the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*. *Chapter 7: Setting up the IBM Netcool/OMNIbus system* covers how to configure the system after installation and how to build and run a standalone ObjectServer. *Chapter 8: Configuring and deploying a multitiered architecture* contains full instructions of how to deploy a multitier IBM Netcool/OMNIbus architecture.

Note: It is recommended to use the Standard Multitier Architecture Configuration to build out ObjectServers, even when deploying just a single failover pair. This ensures that event consistency is maintained in the case of a failover or a failback. If only a single pair of ObjectServers is required, they should be configured as Aggregation ObjectServers. Collection and/or Display ObjectServers can be added later, if needed, depending on load.

IBM Prerequisite Scanner

Prior to installing IBM Netcool/OMNIbus, it is recommended to use the *IBM Prerequisite Scanner* on the intended host machine to ensure that it is a supported platform and that the patch level is adequate for the installation to be successful.

The Prerequisite Scanner is not supplied with IBM Netcool/OMNIbus. You can locate it on the IBM Fix Central web site using the following URL:

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

See the following tech note for information about using the IBM Prerequisite Scanner with IBM Netcool/OMNIbus:

http://www.ibm.com/support/docview.wss?rs=3120&uid=swg21472859

Note: This information is referenced in the section entitled *IBM Prerequisite Scanner* in *Chapter 4: Planning for installation or upgrade* in the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*.

Netcool/OMNIbus WebGUI installation

For full instructions on how to install the IBM Netcool/OMNIbus 8.1 web interface, please refer to *Chapter 6: Installing and updating the WebGUI component* in the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide*. Also refer to the latest IBM Netcool Operations Insight Quick Installation guide available on the IBM Netcool/OMNIbus Best Practices site:

http://ibm.biz/nco_bps

Updating the ObjectServer schema when upgrading

When upgrading Netcool/OMNIbus, it is important to apply the update SQL scripts provided in the `$OMNIHOME/etc` directory in the correct order starting at the incumbent upgrade version up to and including the latest. The upgrade script set includes the following:

```
update70to71.sql
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
update71to72.sql
update72xto73.sql
update73to731.sql
update731to74.sql
update74to74fp3.sql
update74fp3to81.sql
update81to81fp5.sql
update81fp5to81fp7.sql
update81fp7to81fp8.sql
update81fp8to81fp10.sql
```

Note: There may be others added since the time of writing. Please check the `$OMNIHOME/etc` directory for the latest files.

This will ensure that the default ObjectServer internal components are updated to take advantage of new features. Instructions for how to do this are included in the product documentation found here:

http://www.ibm.com/support/knowledgecenter/SSSHTQ_8.1.0/com.ibm.netcool_OMNIbus.doc_8.1.0/omnibus/wip/install/task/omn_upg_obj_schema.html

Note: You will need to apply the update scripts in the correct order from the incumbent system version through to the latest one.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 4 ObjectServers

The IBM Netcool/OMNIbus ObjectServer is the database that holds the event data. It is a proprietary database that uses a client/server middleware for inter-process communication. This chapter contains best practice information that applies to ObjectServer configuration.

ObjectServer properties

This section highlights a number of key ObjectServer properties and provides best practice recommendations for what to set each one to. For a comprehensive listing of all ObjectServer properties, see the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

Connections

The Connections property sets the maximum number of connections that other components can make to the ObjectServer. All components that interact with the ObjectServer consume ObjectServer connections — for example: `nco_sql` sessions, Probe or Gateway connections, IBM Netcool/OMNIbus WebGUI connections or Native Event List connections. Some components such as IBM Netcool/Impact may consume multiple connections depending on how it is configured.

If the ObjectServer reaches its maximum number of connections, no further connections to the ObjectServer are possible until some of the previously made connections are released. The default number of connection for a new ObjectServer is 256. A setting of 1024 is the maximum allowable value, and is recommended as the default setting for all ObjectServers used in a production environment to minimise the chances of an ObjectServer running out of connections.

Granularity

The granularity of the ObjectServer refers to the number of seconds in its internal cycle. The granularity period also dictates how frequently connected clients receive *Insert, Delete, Update and Control* (*IDUC*) information. Once per granularity period, the ObjectServer will also write out to its profiling logs a summary of what has occupied its time during each granularity period — for example: trigger execution and client interaction times. If the total profiling time is approaching the ObjectServer's granularity period, it is an indication that the ObjectServer may be reaching its capacity, and action should be taken to lessen the load.

The default setting for the ObjectServer's granularity is 60 seconds and it is recommended that it be kept at that setting. If more frequent IDUC refreshes are required for a connected client, this should be configured in each respective client's configuration or properties file — for example, in a Gateway's properties file.

Iduc.ListeningPort

By default, the ObjectServer will randomly select an available *ephemeral port* to use for publishing IDUC updates. When an IDUC client connects to the ObjectServer on the primary port, the client queries the ObjectServer for the IDUC port number, and then makes a second connection on that port so that the client can listen out for IDUC notifications.

If IDUC clients are connecting to an ObjectServer where network port restrictions are in place, it can be useful to predefine the number the ObjectServer will select for its IDUC port. This property is used for this purpose. A commonly selected value for this property in IBM Netcool/OMNIbus deployments is `6969`. Although it is a good practice to set this property, it is not necessary to set it if it is not required.

PA.Name, PA.Username, PA.Password

An ObjectServer requires that an accessible Netcool Process Agent is running in order for external procedures to be used. When external procedures are executed, the ObjectServer passes the external

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

action request to the Netcool Process Agent, which then executes the external procedure on behalf of the ObjectServer.

In order for the ObjectServer to be able to communicate with the Process Agent, the following properties need to be set:

- `PA.Name`: the name of the Process Agent as per the interfaces file.
- `PA.Username`: the username of a system user authorised to connect to the Process Agent.
- `PA.Password`: the encrypted password for the system user specified.

The password should be encrypted using the utility: `$OMNIHOME/bin/nco_pa_crypt` (UNIX) or `%OMNIHOME%\bin\nco_pa_crypt.exe` (Windows). See the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more detailed information.

Note: On UNIX platforms where UNIX authentication is used, the username selected for the property `PA.Username` *must* be a member of the `ncoadmin` group. Note that this restriction does not apply where users are authenticated by means of a UNIX *Pluggable Authentication Module* (*PAM*).

Profile

With profiling enabled, the ObjectServer will write profiling information to its log files at the end of each granularity period (default 60 seconds). The overhead required to log this information is negligible, however the information it provides is invaluable particularly during problem determination scenarios in a production environment.

It is a best practice to have profiling enabled (`Profile: TRUE`) for all ObjectServers.

For further reading on ObjectServer profiling, see the section entitled *Best practices for performance tuning* in the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

SecureMode

By default, Probes and Gateways do not have to provide any authentication to the ObjectServer in order to connect. By setting this property to TRUE, Probes and Gateways will have to provide authentication via their respective username and password properties. Passwords should be encrypted using the utility: `$OMNIHOME/bin/nco_pa_crypt` (UNIX) or `%OMNIHOME%\bin\nco_pa_crypt.exe` (Windows). It is recommended to set this property to TRUE for all ObjectServers. See the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more detailed information.

Note: The utility `$OMNIHOME/bin/nco_g_crypt` (UNIX) or `%OMNIHOME%\bin\nco_g_crypt.exe` (Windows) can also be used to encrypt passwords. They produce the same result as `nco_pa_crypt` and `nco_pa_crypt.exe` respectively.

Default triggers

The term *default triggers* refers to the set of triggers that are provided by IBM and exist in the ObjectServer when it is first initialised or created, or any triggers published and released by IBM via the extensions directory. This section highlights a selection of key default triggers within IBM Netcool/OMNIbus and highlights points of note in relation to them.

Do not modify default triggers

Default triggers should never be modified, unless explicitly allowed by official documentation. For example, the *IBM Netcool/OMNIbus 8.1 Installation and Deployment Guide* allows for changes in the `agg_deduplication` trigger to change the way the `Severity` field is updated on deduplication.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The reason for this best practice guideline is that if IBM releases updates to default triggers in the future (eg. in an *interim fix* or *fix pack*), existing custom functionality may be lost when the default triggers are overwritten with the new version — for example: using the `CREATE OR REPLACE` notation. Even then, certain changes may still have to be reapplied, as in the example given above. For this reason, any updates to any default code should be avoided where possible, and thoroughly documented where not avoidable.

The term *default triggers* in this context includes any triggers that exist in the ObjectServer on initial initialisation, the set of triggers provided in the standard multitier architecture, and all triggers provided under the extensions directory. Unless stated otherwise in the documentation, these triggers should not be modified.

If custom functionality is required to augment the default functionality — for example, additional fields requiring update on deduplication — a new, separate reinsert trigger should be created in each case, assigned to a custom trigger group and the field updates included therein. This allows a clear delineation between default, IBM code and custom code, meaning updates to either party's code will not overwrite the other one.

IBM Technical Support fully support both the default and the standard multitier architecture triggers. If an existing default trigger is to be disabled and a replacement custom trigger introduced however, this goes beyond the scope of supportability, and deployment engineers do so at their own risk. The standard multitier architecture configuration has heavy interdependency amongst the configuration components. If default functionality is changed therefore, this may cause other parts to stop functioning correctly.

Although making changes to default functionality is of course allowable if necessitated by the business requirements, it should only be done if absolutely necessary, and extreme care should be taken when doing so. If any trigger logic issues arise because of changing default functionality, IBM will assist in a “best effort” capacity, however the debugging process would be the responsibility of the local engineer.

generic_clear trigger

The generic clear trigger is the standard problem/resolution event correlation trigger in IBM Netcool/OMNIbus. It works by correlating problem events (ie. Type = 1) to resolution events (ie. Type = 2) where the following criteria are met:

Problem event	**Resolution event**
Type = 1	Type = 2
Severity > 0	Severity > 0
Node field matches	
AlertKey fields matches	
AlertGroup field matches	
Manager field matches	
LastOccurrence field timestamp value in problem event is before (ie. less than) value in resolution event	

Where problem events are correlated to resolution events, the problem events are cleared. All resolution events are cleared however, regardless of whether they match any problem events at the time.

Note: A resolution event can potentially match any number of problem events.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The generic clear trigger makes use of a temporary table (`alerts.problem_events`) during its execution. It starts by populating the temporary table with all non-cleared problem events whose `Node`, `AlertKey`, `AlertGroup` and `Manager` match that of a non-cleared resolution event. This incurs a select and a sub-select of the `alerts.status` table. Then for each resolution event in `alerts.status`, it passes over the temporary table marking for clearance any correlated problem events, and clears the current resolution event. This incurs a third select of the `alerts.status` table. Then, it passes over the temporary table one last time and, for each marked problem event in the temporary table, clears the corresponding event in the `alerts.status` table via the Identifier field (direct access — no scan incurred). Finally, it clears out the temporary table in preparation for the next run of the trigger.

This process incurs three scans of the `alerts.status` table and *N + 1* scans of the temporary table, where *N* is the number of active (ie. non-cleared) resolution events. This makes the generic clear trigger's cost-of-operation somewhat fixed regardless of the number of resolution events being processed, because the number of scans of the `alerts.status` table is always limited to three.

Note: It is a best practice to minimise table scans of the `alerts.status` table or any large table wherever possible in trigger design since this operation invariably incurs the biggest load on the ObjectServer.

If a temporary table was not used and the trigger simply cleared all matching problem events in the `alerts.status` table for each non-cleared resolution event present, this would incur *M* scans of the `alerts.status` table, where *M* is the number of non-cleared resolution events. This would mean that the cost-of-operation of the generic clear trigger would go up in proportion to the number of active resolution events present. As the `alerts.status` table grows, this can prove costly in terms of ObjectServer cycles, and consequently limit the scalability of the system.

It is therefore recommended to use the generic clear model to base any custom correlation trigger designs on, and thereby limit the number of scans of the alerts.status table to the minimum possible.

Note: The concept of minimising table scanning should be applied in all trigger or SQL code design. Table scanning happens within any SQL that contains a `where` clause or a `for each row` loop. The use of indexes will dramatically mitigate the cost of such operations however there is a cost incurred in maintaining indexes too, so indexes should be used sparingly. See the section entitled *Indexes* on page 71 for more information.

If generic problem/resolution correlation is desirable in a deployment, an engineer simply must set the relevant fields as per the table above and they will automatically correlate and clear when they reach the Aggregation layer via the generic clear trigger.

The generic clear trigger should only be enabled on Aggregation layer ObjectServers and belong to the `primary_only` group so that it only executes on either the primary or backup Aggregation ObjectServer, not both. Note that this is its default setting in the standard multitier architecture configuration.

GENERIC CLEARING BY DEDUPLICATION

The traditional generic clearing model involves correlating one or more problem events with a resolution event. An alternative to inserting two separate events and then correlating them is simply to clear the original problem event by the incoming event via deduplication. By default, the `Severity` field will always be updated on deduplication. Hence it is possible to clear on deduplication simply by engineering Probe rules files to do so by setting the `Identifier` field to match for both the problem event and the corresponding resolution event.

Note that this only works where the problem to resolution mapping is 1:1. If a many-to-one relationship exists between problem and resolution events, then the traditional generic clearing scheme should be used.

Generic clearing by deduplication is far more efficient than inserting two events and then having to match them up via the traditional generic clearing mechanism. Some business policies however

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

require a separate problem and resolution event, in which case clearing by deduplication is not a usable mechanism.

expire trigger

The `expire` trigger is part of the default trigger group and is responsible for clearing events once they reach their expiry time. The expiry time is set within the `ExpireTime` field and represents an event's time-to-live (in seconds). The `expire` trigger will clear any event where the `LastOccurrence` is more than the `ExpireTime` number of seconds ago. The `delete_clears` trigger will subsequently delete the cleared event two minutes later if no further changes are made to the event (ie. the `StateChange` field is not updated).

This trigger should only be enabled on Aggregation layer ObjectServers. The standard multitier architecture configuration disables the default expire trigger in Collection layer ObjectServers and creates an alternative trigger called `col_expire`, since it needs to consider whether or not it has been forwarded up to the Aggregation layer. It does not expire events that have not been forwarded to the Aggregation layer yet.

Note: It is a best practice to ensure that every event has an `ExpireTime` value set. No events should be allowed to remain indefinitely in the ObjectServer. See the section entitled *ObjectServer housekeeping* on page 82 for further discussion on this topic.

---

delete_clears trigger

The `delete_clears` trigger is part of the default trigger group and is responsible for deleting cleared events. Events will be deleted if their `Severity` field is set to 0 (ie. clear) and they have not been modified for more than 120 seconds (ie. when the `StateChange` timestamp field is older than two minutes ago).

It is recommended to configure custom triggers to clear events in the ObjectServer — and then let the `delete_clears` trigger remove them. This allows any connected clients (such Event List users or any other read client) a chance to see and register the event clearing, before the event disappears from their view. There may be circumstances however where immediate deletion is preferred — for example when managing events that affect correlation outcomes.

This trigger should only be enabled on Collection and Aggregation layer ObjectServers. This is set up for you when using the standard multitier architecture configuration.

clean_details_table trigger

The `clean_details_table` trigger is part of the `default_triggers` group (or `dsd_triggers` group on Display ObjectServers) and is responsible for removing rows from the `alerts.details` table where there is no longer any corresponding row in the `alerts.status` table.

This trigger should *always* be enabled on *all* ObjectServers.

clean_journal_table trigger

The `clean_journal_table` trigger is part of the default trigger group (or `dsd_triggers` group on Display ObjectServers) and is responsible for removing rows from the `alerts.journal` table where there is no longer any corresponding row in the `alerts.status` table.

This trigger should *always* be enabled on *all* ObjectServers.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Custom trigger groups

If custom triggers are to be implemented and deployed, one or more custom trigger groups should be created to accommodate them. This best practice helps Netcool solution developers to create a clear delineation between default IBM code and any customisations.

This separation of default IBM code from customisations is important to ensure that if changes are made to default IBM code — for example, a fix pack that implements a change to a default trigger — it won't affect the functionality provided by any customisations and vice versa.

EXAMPLE:

Widgetcom has a requirement to implement a number of custom triggers. The Netcool Administrator elects to create a custom group within the ObjectServer, and creates the following SQL file to apply to the Aggregation ObjectServers. Any custom triggers that are created from this point on will be assigned to this custom trigger group.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
------------------------------------------------------------------
-- CREATE A CUSTOM TRIGGER GROUP FOR WIDGETCOM
CREATE TRIGGER GROUP widgetcom_triggers;
go

-- ENABLE THE CUSTOM TRIGGER GROUP
ALTER TRIGGER GROUP widgetcom_triggers SET ENABLED TRUE;
go
```

Custom triggers

This section outlines general best practice guidelines for the construction of custom ObjectServer triggers. All custom functionality delivered in the form of triggers should be included in the solution delivery SQL file. For further information on solution delivery, see the section entitled *Solution delivery* on page 35.

Note: Also see the section entitled: *Best practices for creating triggers* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more tips on trigger creation.

Temporal triggers

Temporal triggers are used to carry out ObjectServer automations on a timed basis — for example: *"every 300 seconds..."*. This section provides an example business requirement, and provides a sample solution using a temporal trigger.

EXAMPLE:

Widgetcom has a requirement to periodically downgrade the severity of non-acknowledged warning and minor events based on their age (`FirstOccurrence`). Minor events should be downgraded to warning events once they are an hour old, and warning events should be cleared once they are two hours old. The Netcool Administrator elects to provide this functionality via a temporal trigger.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
------------------------------------------------------------------
-- CREATE THE NEW EVENT SEVERITY DOWNGRADE TRIGGER
CREATE OR REPLACE TRIGGER downgrade_events
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER downgrade_events
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This trigger periodically checks the age of minor and warning
events and downgrades them as they age past thresholds.  Minor
events get downgraded to warning events once they have been in the
ObjectServer unacknowledged for 1 hour.  Warning events get cleared
once they have been in the ObjectServer unacknowledged for 2 hours.'
EVERY 301 SECONDS
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
declare
   now utc;

begin
   -- INITIALISE VARIABLE
   set now = getdate();

   -- CHECK UNACKNOWLEDGED WARNING AND MINOR EVENTS
   for each row downgrade in alerts.status where
         downgrade.Acknowledged = 0 and
         downgrade.Severity in (2,3)
   begin

         -- DOWNGRADE MINOR EVENTS IF MORE THAN 1 HOUR OLD
         if (downgrade.Severity = 3 and
               (downgrade.FirstOccurrence < (now - 3600))) then

               set downgrade.Severity = 2;

         -- CLEAR WARNING EVENTS IF MORE THAN 2 HOURS OLD
         elseif (downgrade.Severity = 2 and
               (downgrade.FirstOccurrence < (now - 7200))) then

               set downgrade.Severity = 0;
         end if;
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
   end;
end;
go
```

NOTES:

- The trigger header has a row of hyphens to clearly demarcate it from the previous SQL file entry, plus a comment that describes what this next section of code does.
- The syntax to create the trigger includes the key words OR REPLACE. This allows for the reapplication of the SQL file against the ObjectServer and subsequent recreation of the trigger. Without these key words, the trigger must first be removed before it can be recreated. An attempt to create a trigger that already exists without using the key words OR REPLACE will result in an error. It is therefore recommended to use CREATE OR REPLACE over CREATE on its own.
- The new trigger is assigned to *Widgetcom's* new custom trigger group.
- The comment section contains a title, last modified section, including the name of the creator, their role, and the date, as well as a full description of the trigger and what it does in plain English. Do not use abbreviations or jargon in comments. Comments sections should be very clear and thorough in their description, and leave the reader in no doubt as to what the trigger does and how.
- The trigger is configured using the WHEN clause to only execute if the ObjectServer is the current acting primary ObjectServer. This is usually appropriate when installing custom temporal triggers on Aggregation layer ObjectServers. This is not appropriate when installing custom temporal triggers on Collection layer ObjectServers however because Collection layer ObjectServers work independently of each other and so triggers should always be active.
- Temporal trigger poll times should be prime number values. This minimises the possibility of two temporal triggers firing at the same time, thereby helping to even out the load on the ObjectServer.

Note: When two temporal triggers happen to fire at the same time, they will be executed sequentially first in order of priority and second in ascending alphabetical order by name.

- Although the trigger has a detailed comments section at the top, the trigger code also contains clear, descriptive comments in-line in the code. This is essential for the reader to identify what each line of series of lines of code does.
- To simplify the code, the function `getdate()` is only run once, the result saved into a variable and the variable used instead. A variable initialisation section is included at the start of the trigger code so that initial values can be easily determined by the reader.
- The FOR EACH ROW loop has the WHERE clause items ordered by increasing cost of execution. See the section entitled *Ordering of SQL where clauses* later in this chapter for more detail on this topic.
- The trigger code indentation conventions and use of whitespace are clear and consistent throughout. This is more professional and makes for easier reading of the code.

Database triggers

Database triggers are used to carry out ObjectServer automations based on a change to a specified ObjectServer table — for example, when an insert, reinsert, update to, or deletion from a table occurs. This section provides an example business requirement — and provides a sample solution using a database trigger.

Note that it is allowable to have multiple database triggers that fire on the same action — for example: on insert into the `alerts.status` table. The order in which database triggers fire for any given action is determined by their PRIORITY setting. The standard multitier triggers are all configured to run at priority 2, so that custom triggers can be created with a priority of 1 and execute first.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: Where two database triggers have the same priority level, the order that they will run is indeterminable. If the firing order is important therefore, the priorities should be set accordingly.

When debugging multiple database triggers that fire for the same action, note that the values stored in the `new.*` and `old.*` variables may be modified by any of the triggers. Secondary triggers will inherit the values set by triggers that fire first. Dependency on the values of variables therefore as well as the firing order of the triggers both have to be carefully considered when “stacking” triggers to fire sequentially.

EXAMPLE:

Widgetcom has a requirement to increase the severity of an event if it recurs within 10 minutes since the previous occurrence. This should only apply to events whose Class value is 99999. If an event is already critical (ie. has a severity of 5), the severity should be left unchanged. The Netcool Administrator elects to provide this functionality via a database trigger.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
------------------------------------------------------------------
-- CREATE THE NEW EVENT SEVERITY UPGRADE TRIGGER
CREATE OR REPLACE TRIGGER upgrade_events
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER upgrade_events
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This trigger acts only on non-critical Class = 99999 events.  If
this recurrence of the event is less than 10 minutes since the
last one, it increases the Severity by 1.'
BEFORE REINSERT ON alerts.status
FOR EACH ROW
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
begin

   -- CHECK IF CLASS IS 99999, EVENT IS NON-CRITICAL AND THAT
   -- LASTOCCURRENCE FIELD IS LESS THAN 10 MINUTES OLD
   if (old.Class = 99999 and
         old.Severity < 5 and
         old.LastOccurrence > (getdate() - 600)) then

         set new.Severity = old.Severity + 1;
   end if;
end;
go
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Signals and signal triggers

There are a number of actions defined within the ObjectServer that raise a *Signal* when that action occurs. For example, when a client connects to the ObjectServer or a Gateway finishes its resynchronisation, a signal will be raised. A signal trigger can then be constructed that will fire on the specified signal. For example, a trigger could be constructed to write to a log file whenever a user logs in.

In addition to the default, built-in signals, custom signals can also be created. Signal triggers can be constructed to fire when these signals are raised also. Custom signals are raised either manually by a user or automatically within a trigger or procedure via the `RAISE SIGNAL` command.

EXAMPLE:

Widgetcom have two states of operation within their NOC: “STORM” — when an event storm condition is declared by the NOC Administrator and “NORMAL” for all other times. When the current state changes from one to the other, a number of tasks need to be carried out on the Aggregation ObjectServers. When an event storm condition is declared, a special event pruning trigger (`prune_events`) should be enabled and an alert e-mail sent to the head of operations. When the event storm condition is over, the event pruning trigger should be disabled, and another e-mail sent to the head of operations advising them of the fact. The NOC Administrator would like to be able to have a single SQL command to run to toggle between these two states. The Netcool Administrator elects to provide this functionality via the creation of a custom signal and a signal trigger.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
------------------------------------------------------------------
-- CREATE THE EVENT STORM CUSTOM SIGNAL
CREATE OR REPLACE SIGNAL event_storm_signal
(storm_status char(6))
COMMENT 'SIGNAL event_storm_signal
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This signal is raised by the NOC Administrator to indicate when
it is declared that an event storm condition is beginning or
ending.  It is used in conjunction with the signal trigger
event_storm_toggle.  Input parameters should be either STORM or
NORMAL.';
go

------------------------------------------------------------------
-- CREATE THE EVENT STORM SIGNAL TRIGGER
CREATE OR REPLACE TRIGGER event_storm_toggle
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER event_storm_toggle
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This trigger is fired when the custom signal event_storm_signal is
raised by the NOC Administrator to indicate when it is declared
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
that an event storm condition is beginning or ending.  The signal
will be called with one argument - either STORM or NORMAL.
In the event of a STORM call, the pruning trigger prune_events is
enabled and the head of operations is notified.  In the event of
a NORMAL call, the pruning trigger prune_events is disabled and the
head of operations notified.'
ON SIGNAL event_storm_signal
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
begin

   -- IF VALUE OF INCOMING STATUS IS STORM
   if (%signal.storm_status = 'STORM') then

         -- ENABLE EVENT STORM PRUNING TRIGGER
         alter trigger prune_events set enabled true;

         -- SEND NOTIFICATION EMAIL TO OPERATIONS HEAD
         execute send_email(
               'Netcool System',
               5,
               'Netcool Storm started',
               'operations_head@widgetcom.com',
               'An IBM Netcool event storm condition has been
declared.',
               'netcoolhost');

   -- IF VALUE OF INCOMING STATUS IS NORMAL
   elseif (%signal.storm_status = 'NORMAL') then

         -- DISABLE EVENT STORM PRUNING TRIGGER
         alter trigger prune_events set enabled false;

         -- SEND NOTIFICATION EMAIL TO OPERATIONS HEAD
         execute send_email(
               'Netcool System',
               2,
               'Netcool Storm finished',
               'operations_head@widgetcom.com',
               'An IBM Netcool event storm condition has ended.',
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
                'netcoolhost');
   end if;
end;
go
```

The NOC Administrator should be advised when the new signal is installed and informed of its correct usage is as follows:

```
RAISE SIGNAL event_storm_signal STORM;
go
```

...or:

```
RAISE SIGNAL event_storm_signal NORMAL;
go
```

NOTES:

- Instead of creating multiple custom signals for different variations of the same event, rather create one signal that can accept an input parameter.
- This example solution makes use of the default external procedure `send_email` which in turn makes use of the default utility shell script `nco_mail`.
- The NOC Administrator signal commands can be run from an SQL prompt — or could be incorporated into a restricted Event List tool for easier execution.

Using the WHEN clause

When creating a custom trigger, consideration should be given as to which ObjectServer tier the new trigger will run on, and whether the trigger should be enabled to run all the time or only on the acting primary.

If a trigger will be installed and run on a Collection layer ObjectServer, it can typically be left to run all the time, requiring no specific modification. Typically however, temporal triggers installed on Aggregation layer ObjectServers should only be active on the acting primary — for example: on the primary Aggregation ObjectServer or the backup Aggregation ObjectServer when the primary is down. In such cases, a WHEN clause should be added to the trigger used to restrict its execution.

An example WHEN clause that will restrict execution to an acting primary is provided below and should be added to the trigger in the appropriate position within the solution delivery file:

```
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
```

This addition will restrict a trigger's execution to only run when the ObjectServer it is running on is the current primary. This is important to ensure that events are only modified in one place at a time and thereby avoid race conditions.

Note: See the section entitled *Temporal triggers* earlier in this chapter for an example of the use of the WHEN clause in this context.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Use of comments

The use of comments within the context of programming is a fundamental best practice of software design. It is also a best practice of solution development within the Netcool suite. Populating the comments sections and putting comments in-line in the code is a fundamental best practice, and one which is often neglected in IBM Netcool/OMNIbus deployments.

If an IBM Netcool configuration is thoroughly commented, an engineer beginning work on an existing system can deduce very quickly what a piece of functionality is supposed to do, and how it is supposed to do it.

For the ObjectServer, extensive commenting should be present in all triggers, procedures, signals, tools (*Description* field) and anywhere else it makes sense to include them. This section contains a number of best practice conventions which should be followed wherever possible.

MAKE SURE THE COMMENT/DESCRIPTION BOX IS COMPLETED

For triggers and signals, there is a *Comment* box available, with a *Description* box for tools. These should always be populated and include at the very least:

- The creator and last modifier of the trigger/signal/tool;
- The date of creation and/or the date of last modification;
- A thorough description of what the trigger/signal/tool does and how it does it.

Note: Default out-of-the-box comment sections do not include creator or date information.

INCLUDE IN-LINE COMMENTS

In-line comments must be entered for all components that contain lines of code (ie. tools, triggers, procedures, SQL files). It is better to have too much in-line commenting that not enough. See the trigger examples in the previous few sections for examples of in-line commenting.

MAKE IN-LINE COMMENTS STAND OUT

To make your code more readable, you should make your comments stand out from the lines of code. In the example triggers in the previous section, this is done by capitalising the comments. Additional break lines can also be added to separate blocks of code. In the example triggers in the previous section, this is done by placing rows of hyphens between blocks of code.

STANDARDISE YOUR COMMENT LAYOUT AND FORMATTING

Whatever conventions you use for the formatting of your comments, you should make it consistent throughout your code to make it more readable and of a more professional appearance. In the trigger examples given in the previous sections, each comment header follows the same format and all in-line comments are in capital letters.

Standardisation of layout and formatting

To improve the readability and professionalism of code, Netcool engineers should adopt a consistent approach to code layout and formatting. Decide what your conventions will be and keep layout and formatting uniform throughout your custom content.

Using the temporal trigger in a previous section as an example, the following highlights some conventions that have been used an applied. Note that the following conventions are *suggested* conventions only. It matters less what conventions are adopted, but more that the conventions that are adopted are followed *consistently*.

THE HEADER

```
1 ----------------------------------------------------------------
2 -- CREATE THE NEW EVENT SEVERITY DOWNGRADE TRIGGER
3 CREATE OR REPLACE TRIGGER downgrade_events
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
 4 GROUP widgetcom_triggers
 5 PRIORITY 1
 6 COMMENT 'TRIGGER downgrade_events
 7 Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
 8 This trigger periodically checks the age of minor and warning
 9 events and downgrades them as they age past thresholds.  Minor
10 events get downgraded to warning events once they have been in
11 the ObjectServer unacknowledged for 1 hour.  Warning events get
12 cleared once they have been in the ObjectServer unacknowledged
13 for 2 hours.'
14 EVERY 301 SECONDS
15 WHEN get_prop_value('ActingPrimary') %= 'TRUE'
```

The following notes apply to the header section above:

- Line 1: A horizontal line of hyphen characters are used to demarcate each solution delivery SQL file item. This improves readability. Any row of characters can be used if they are preceded by two hyphen characters, which designates the line as a comment.
- Line 2: The trigger is headed by a one-line comment saying what the trigger is for.
- Line 3: The keywords are all capitalised while the content words are all in lower case.
- Line 3: The trigger includes the clause “OR REPLACE” to allow for reapplication of the SQL file if updates are made to the solution delivery file and it needs to be reapplied.
- Lines 3 to 14: Each trigger header parameter is on a separate line to aid readability.
- Lines 6 to 13: The comment section has the type of object and the name on the first line, a “last modified by” line on the second line and the description on subsequent lines.

THE DECLARE SECTION

```
16 declare
17       now utc;
18
```

The following notes apply to the declare section above:

- Line 16 onwards: From the keyword “declare” onwards, all code is in lowercase. This is because everything below and including this point appears in the *Action* tab when viewing the trigger from within the Netcool Administrator tool. This aids readability since it will contrast well with any in-line comments — which are in capitals.
- Line 17: The local variable names (eg. `now`), when present, are indented by one tab character and separated from the type (ie. `utc`) by a single space character.
- Line 18: A blank line is left intentionally to separate the declare section from the main body of the trigger code to aid readability.

THE VARIABLE INITIALISATION SECTION

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
19 begin
20      -- INITIALISE VARIABLE
21      set now = getdate();
22
```

The following notes apply to the variable initialisation section above:

- Line 19 onwards: After the BEGIN statement, all subsequent lines are indented by at least one tab character. This will continue until the final END statement that closes the trigger.
- Line 20: The variable initialisation section is clearly indicated with a comment.
- Line 21: Variables are created with meaningful names.
- Line 21: If a function is called more than once throughout the code block (eg. `getdate()`), consider using a variable to hold the result and reuse the variable instead. This can simplify the appearance of code, improving its readability.
- Line 22: A blank line is left intentionally to separate the variable initialisation section from the next section of the trigger code to aid readability.

THE "FOR EACH ROW" LOOP CONSTRUCT HEADER

```
23      -- CHECK UNACKNOWLEDGED WARNING AND MINOR EVENTS
24      for each row downgrade in alerts.status where
25            downgrade.Acknowledged = 0 and
26            downgrade.Severity in (2,3)
27      begin
28
```

The following notes apply to the for-each-row construct header above:

- Line 23 onwards: The indentation continues consistent with the previous sections, namely being indented by one tab space.
- Line 23: The FOR EACH ROW construct is headed by a comment that specifies what the loop construct is looking for. It may also include a fuller explanation of what it is for, if needed.
- Line 24: The variable selected for the loop construct has a meaningful name within the context of the loop. In this case, the loop is examining candidate events for downgrading, hence the name `downgrade`.
- Lines 24 to 26: The FOR EACH ROW statement is included in full on the first line up to and including the WHERE keyword. Any WHERE clause conditional terms are itemised on the lines following, one per line and indented by a further tab character since they are sub-elements of the FOR EACH ROW statement, and to aid readability.
- Lines 25 and 26: The WHERE conditional terms are ordered in increasing order of cost-of-execution. In this case, a single integer comparison is first, followed by an integer comparison of a variable within a set of values.

Note: The topic of the ordering of SQL WHERE clauses is covered later in this document in the section entitled *Ordering of SQL where clauses*.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- Line 27: The begin statement drops back to the same level of indentation as the FOR EACH ROW loop to help demarcate the code that will be contained within the loop.
- Line 28: A blank line is left intentionally to separate the FOR EACH ROW loop header from the loop contents to aid readability.

THE IF STATEMENT CONDITIONAL CONSTRUCT HEADER AND CONTENTS

```
29              -- DOWNGRADE MINOR EVENTS IF MORE THAN 1 HOUR OLD
30              if (downgrade.Severity = 3 and
31                    (downgrade.FirstOccurrence < (now - 3600))) then
32
33                    set downgrade.Severity = 2;
34
```

The following notes apply to the conditional construct header and contents above:

- Line 29: The IF statement is preceded by a comment that explains what is checking for. If the resulting action is simple, it can be incorporated here also, otherwise it should appear in-line below.
- Lines 30 and 31: The conditional terms are ordered in increasing order of cost-of-execution. In this case, a single integer comparison is executed before the calculation with respect to the `FirstOccurrence` field.
- Lines 30 and 31: As for the FOR EACH ROW construct, if there are more than one conditional terms, they appear one per line, indented, to aid readability.
- Line 32: A blank line is left intentionally to separate the IF statement header from the IF statement contents to aid readability.
- Line 33: The IF statement contents are indented in one tab space more than the IF statement header.
- Line 34: A blank line is left intentionally to separate the IF statement contents from the next code block to aid readability.

THE ELSEIF STATEMENT CONDITIONAL CONSTRUCT HEADER AND CONTENTS

```
35              -- CLEAR WARNING EVENTS IF MORE THAN 2 HOURS OLD
36              elseif (downgrade.Severity = 2 and
37                    (downgrade.FirstOccurrence < (now - 7200))) then
38
39                    set downgrade.Severity = 0;
40              end if;
```

The following notes apply to the ELSEIF conditional construct header and contents above:

- Line 35: As in the previous example of the IF statement, the ELSEIF statement is preceded by a comment that explains what it is checking for and is at the same indentation level.
- Line 40: The END IF statement is at the same indentation level as the IF and ELSEIF statements.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

THE FOOTER

```
41       end;
42 end;
43 go
```

The following notes apply to the footer section above:

- Lines 41 and 42: The closing END statements should align with their respective opening counterparts in terms of indentation. Consistent indentation also helps you to find any syntax errors that may be present. If an engineer gets to the end of the component they are working on and the indentation is not realigned to the margin, the chances are a closing END statement has been missed.
- Line 43: An SQL file will normally contain a GO statement under each component or code block. This causes the previous line or lines to be executed. By placing GO statements after each component or code block, it makes it easier to narrow down the location within your solution delivery file of any syntax errors that may be present in your code.

Avoid table scanning where possible

With respect to writing ObjectServer SQL, a statement that involves a table scan is invariably the costliest operation there is. A table scan is incurred from an SQL statement any time a WHERE clause is used.

Examples include:

```
update alerts.status set Acknowledged = 1 where Location = 'London';
delete from custom.mytable where IDField < 2;
for each row myrow in alerts.status where myrow.Node like 'aaa' ...
```

Whenever possible, avoid table scanning in any ObjectServer SQL. If it is necessary to do so, ensure that the search terms in the where clause are ordered in increasing execution cost. See the next section entitled *Ordering of SQL where clauses* in this document for more information about the relative cost of execution with respect to ObjectServer SQL.

Note: When just a direct match is made with either a primary key field or any indexed field within a WHERE clause, a table scan is not incurred and the WHERE clause will be relatively efficient. Primary key fields are automatically indexed on creation by default. See the section entitled *Indexes* in this document for further reading on the types of index and examples of each type's application.

Avoid nesting for-each-row loops

A FOR EACH ROW loop is a convenient tool for performing an action for each event that matches a given *where* clause. Although relatively costly in terms of ObjectServer processing cycles, when used responsibly and with an efficient *where* clause, it can be a powerful tool for implementing custom requirements.

A FOR EACH ROW construct invokes a scan of the table it is applied against. Such actions are typically the most expensive type of SQL action in a trigger, particularly as the size of the table it is scanning increases.

For-each-row loops should be used sparingly, therefore, and care should be taken not to use the for-each-row construct when a more efficient construct could be used instead.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

A particularly expensive example of for-each-row construct usage to avoid is where they are nested one within the other. When for-each-row loops are nested, it *multiplies* the number of scans of the alerts.status table incurred, by the number of rows matched in the outer loop.

Note: The exception to this advice would be where both for-each-row loops use only indexed fields. That said, performance testing should be carried out on the trigger to validate its performance is to an acceptable level.

EXAMPLE:

Widgetcom have a temporal trigger that carries out the following action:

```
for each row escrow in alerts.status where
   escrow.Escalate = 1 and
   escrow.Severity > 3
begin

   for each row majrow in alerts.status where
         majrow.Escalate = 1 and
         majrow.Severity = 4
   begin
         set majrow.Escalate = 2;
   end;

   for each row critrow in alerts.status where
         critrow.Escalate = 1 and
         critrow.Severity = 5
   begin
         set critrow.Escalate = 3;
   end;
end;
```

This poorly written trigger unnecessarily incurs 1 + 2N scans of the alerts.status table, where N is the number of rows in the ObjectServer in which the Escalate field is set to 1 and the Severity is either Major (ie. 4) or Critical (ie. 5).

This trigger is potentially extremely costly and likely to severely impact the performance of the ObjectServer as the number of matching rows increases as well, as the overall number of resident events.

The Netcool Administrator re-writes the trigger as follows:

```
for each row escrow in alerts.status where
   escrow.Escalate = 1
begin
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
   if (escrow.Severity = 4) then

         set sevrow.Escalate = 2;

   elseif (escrow.Severity = 5) then

         set escrow.Escalate = 3;

   end if;
end;
```

This version of the trigger incurs just one scan of the alerts.status table regardless of the number of rows that match the WHERE clause. Moreover, if a match is found in the first IF statement, it drops through to the bottom and does not evaluate the second IF statement; thanks to the use of the IF-ELSEIF construct.

Ordering of SQL where clauses

When writing ObjectServer SQL, care should be taken to make it as efficient as possible; including the ordering of WHERE clauses. As discussed in the previous section, a WHERE clause incurs a scan of the table concerned. Although the ObjectServer has a query optimiser built-in, it optimises on a “best effort” basis and consequently is not perfect. Hence constructing efficient code is still considered a best practice.

The following field types are available within ObjectServer tables and are listed 1 to 3 in increasing order of cost of comparison.

1. Boolean, Integer, Integer64, Incr, UTC, Unsigned, Unsigned64 (all integers)
2. Real
3. Char, VarChar (all strings)

Integers are the cheapest in terms of cost of comparison in WHERE clauses, followed by real numbers and finally by strings.

Note: The ObjectServer query optimiser treats integers and real numbers the same in terms of cost of comparison. In reality, integer comparisons are marginally cheaper; therefore integer comparisons should be put ahead of real comparisons when manually constructing WHERE clauses in your code.

The list below expands on the ObjectServer field types to provide relative costs of execution for ObjectServer WHERE clauses in increasing order.

1. True/False (Boolean)
2. Integer comparison (=, !=, <, >, etc.)
3. Real comparison (=, !=, <, >, etc.)
4. String match (=, !=)

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

5. String complex comparison (LIKE, NOT LIKE, %<, %>, etc.)
6. Integer ANY/ALL/IN
7. String ANY/ALL/IN
8. Subselect - that is a nested SELECT statement

Note: The above order provides a general guideline and reflects the ordering the ObjectServer query optimiser uses. It is important to note however that a particularly complex LIKE string comparison (ie. a regular expression match) can potentially be costlier than a simpler IN list comparison. The above ordering also does not consider the makeup of the data contained in the ObjectServer, as this can make one type of comparison work out cheaper than another type of comparison. It is advisable that benchmark testing is carried out on the different permutations of WHERE clause ordering to find the most efficient in your environment. Note that this exercise is only worth doing for queries that take a long time to run and/or are executed regularly, and not for all WHERE clauses. Generally speaking, the above ordering will suffice.

EXAMPLE:

A *Widgetcom* engineer has constructed the following SQL update statement for inclusion in a trigger:

```
update alerts.status set Acknowledged = 1 where
   Node like 'router' and
   LocationID in (1, 2, 4) and
   Location = 'LONDON' and
   Severity > 3;
```

The Netcool Administrator reorders the SQL query to the following:

```
update alerts.status set Acknowledged = 1 where
   Severity > 3 and
   Location = 'LONDON' and
   Node like 'router' and
   LocationID in (1, 2, 4);
```

The *integer* comparison is done first, followed by the *string match*, then the *string complex comparison* and finally, the *integer in* comparison.

Note: For further reading, see the section entitled *Best practices for creating triggers* in the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

Use of the Evaluate tab

When a trigger contains an EVALUATE clause, a temporary table is created in memory to hold the results of the SELECT statement. The amount of time taken to construct, and the memory resources consumed depends on the number of columns being selected, and the number of rows matched by the condition in the WHERE clause.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Generally, use of the EVALUATE clause is relatively inefficient and its use should be avoided whenever possible. In most cases, you can replace an EVALUATE clause with a FOR EACH ROW clause, which cursors over the data and does not incur the overhead of creating a temporary table.

Note: See the section entitled *Avoid the EVALUATE clause* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more information on the EVALUATE tab.

There are scenarios however where the use of an EVALUATE clause is necessary, for example when a GROUP BY clause is applied to an SQL query.

EXAMPLE:

Widgetcom has a requirement to write out to a log file once an hour a list of the *unique* locations (ie. `Location` field) present in the ObjectServer's events. Each location should only appear once in the log file, regardless of how many events there are with each location.

The Netcool Administrator decides to provide this functionality via a temporal trigger that uses a GROUP BY clause within the EVALUATE section.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-----------------------------------------------------------------
-- CREATE LOG FILE HANDLE FOR WRITING NODE NAMES TO
CREATE OR REPLACE FILE location_log
'/opt/IBM/tivoli/netcool/omnibus/log/node_log.log'
MAXFILES 3
MAXSIZE 1 MBYTES;
go

-----------------------------------------------------------------
-- CREATE TRIGGER TO LOG NODES TO FILE
CREATE OR REPLACE TRIGGER write_node_to_log
GROUP widgetcom_triggers
ENABLED TRUE
PRIORITY 1
COMMENT 'TRIGGER write_node_to_log
Last modified by: J Smith (jsmith) - Netcool Admin - 24-Aug-2011
Once an hour, this trigger writes out to a log file a list of all
event locations currently present in the ObjectServer.  Each
location is only written to the log file once - regardless of how
many events have that location.'
EVERY 1 HOURS
EVALUATE
SELECT Location FROM alerts.status
   GROUP BY Location
   ORDER BY Location ASC
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
BIND AS all_locations
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
begin

   -- WRITE A LOG FILE HEADER
   write into location_log values
         ('----------------------------');
   write into location_log values
         ('LOCATIONS AS AT ', to_char(getdate()));

   -- PARSE OVER THE SET OF LOCATIONS
   for each row this_location in all_locations
   begin

         -- WRITE EACH ONE TO THE LOG FILE
         write into location_log (this_location);
   end;
end;
```

NOTES:

- On Windows platforms, the filename will be in the form: `C:\Program Files\IBM\tivoli\netcool\omnibus\log\node_log.log`.
- The performance of triggers that use the "WRITE INTO" SQL command should be carefully monitored as they may cause bottlenecks depending on how much work they must do. File I/O operations can be relatively costly.

Default escalations

There are a number of default escalation components present in the ObjectServer. This section outlines their purpose and provides some guidelines for their use.

SuppressEscl

The purpose of the `SuppressEscl` field is to provide a default field as a basis for the suppression or escalation of events. Although there are no default triggers that carry out any actions on events based on this field, it is provided as a framework on which customised actions can be built. It is recommended to use this field as a basis for any event escalation customisations that are implemented.

The suppression level of an event or group of events can be manually set by operators via an Event List tool or via custom triggers. Some default triggers are also configured to modify the `SuppressEscl` field, including `flash_not_ack` and `escalate_off`.

The following list details the different escalation values and what state of escalation each value represents:

```
0: Normal
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
1: Escalated
2: Escalated-Level 2
3: Escalated-Level 3
4: Suppressed
5: Hidden
6: Maintenance
```

Note: The `SuppressEscl` field is an integer field. Conversions are present in the ObjectServer that translate the integer values to meaningful textual values in the Event List.

flash_not_ack

When an event has the `Flash` field set to 1, it enables Event List functionality where the event will toggle between its normal colour and a darker shade of the same colour. This gives the appearance of a light blinking on and off and is called *flashing*. The purpose of this blinking behaviour is used to draw an operator's attention to an event that requires attention.

This trigger is a default trigger that enables flashing (`Flash = 1`) and escalates events (`SuppressEscl = 1`) that are more than 10 minutes old, are of critical severity (`Severity = 5`) but which have not yet been acknowledged by a user (`Acknowledge = 0`).

This trigger is disabled by default and must be manually enabled if required. This can be done by adding the following to your solution delivery SQL file:

```
-------------------------------------------------------------------
-- ENABLE flash_not_ack TRIGGER
ALTER TRIGGER flash_not_ack SET ENABLED TRUE;
go
```

escalate_off

This is a default trigger that disables flashing (`Flash = 0`) and de-escalates events (`SuppressEscl = 0`) that have previously had flashing enabled (`Flash = 1`) but are now either acknowledged (`Acknowledged = 1`) or if the event has now been cleared (`Severity = 0`). This trigger works in conjunction with the default trigger *flash_not_ack*.

This trigger is disabled by default and must be manually enabled if required. This can be done by adding the following to your solution delivery SQL file:

```
-------------------------------------------------------------------
-- ENABLE escalate_off TRIGGER
ALTER TRIGGER escalate_off SET ENABLED TRUE;
go
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

mail_on_critical

This is a default trigger that sends an e-mail about critical alerts that are unacknowledged (`Acknowledged = 0`), are more than 30 minutes old and that haven't already had an e-mail sent out (`Grade < 2`).

This trigger uses the `Grade` field as a flag, which is by default used to indicate an event's state of escalation (0: not escalated, 1: escalated). It sets the `Grade` field to 2 after it has sent the first e-mail for each event so that it doesn't send duplicate e-mails every time the trigger fires. (The trigger only acts on events whose `Grade` field value is less than 2.)

It makes use of a default external procedure `send_email`, which, in turn, executes a default utility script `$OMNIHOME/utils/nco_mail`.

Note: This escalation is UNIX specific due to the preset configuration of the `send_email` procedure. If this is required on a Windows platform, a new Windows-specific external procedure should be created using `send_email` as a model. Default procedures should not be modified wherever possible.

This trigger is disabled by default and must be manually enabled if required. This can be done by adding the following to your solution delivery SQL file:

```
------------------------------------------------------------------
-- ENABLE mail_on_critical TRIGGER
ALTER TRIGGER mail_on_critical SET ENABLED TRUE;
go
```

Procedures

ObjectServer *SQL* procedures are analogous to the procedure construct found in many programming languages. They allow a piece of code to be “packaged” up and reused; either with or without input or output parameters. An ideal application of an SQL procedure is where the same SQL action needs to be carried out for a number of different triggers. By putting the code into an SQL procedure and calling it from multiple places, it reduces the overall number of lines of code, as well as reducing the maintenance overheads should a change subsequently need to be made to the SQL action.

External procedures are used to execute an external action on the host system — for example, an executable script — via the Netcool Process Agent.

ObjectServer procedures are precompiled at the moment they are created or saved, allowing for faster execution when they are run.

SQL procedures

As mentioned above, SQL procedures are used to package up fragments of ObjectServer SQL that are used repetitively within the ObjectServer to make for fewer lines of code and to simplify maintenance overheads.

EXAMPLE:

Widgetcom has a number of triggers that carry out tasks based on the number of critical events in the ObjectServer. These triggers each contain a FOR EACH ROW loop that counts the number of critical events in the ObjectServer, so it can deduce whether or not to carry out the allotted task.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Netcool Administrator decides to create an SQL procedure to store this code to reduce the overall lines of code contained within the trigger set and to allow for faster execution.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-----------------------------------------------------------------
-- CREATE PROCEDURE TO COUNT UP THE NUMBER OF CRITICAL EVENTS
CREATE OR REPLACE PROCEDURE count_up_criticals (
   OUT count_of_criticals INTEGER )
declare my_counter integer;
begin

   -- PROCEDURE count_up_criticals
   -- Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
   -- This procedure counts up the number of critical events in the
   -- ObjectServer and returns them in the variable named
   -- count_of_criticals.
   --
   -- Usage: execute procedure count_up_criticals(my_integer_var);
   --

   -- INITIALISE COUNTER
   set my_counter = 0;

   -- ITERATE OVER THE alerts.status TABLE KEEPING
   -- A COUNT OF CRITICAL EVENTS
   for each row this_event in alerts.status where
         this_event.Severity = 5
   begin

         -- INCREMENT THE COUNTER FOR EACH CRITICAL EVENT FOUND
         set my_counter = my_counter + 1;
   end;

   -- SET RETURN VARIABLE TO THE CALCULATED TOTAL COUNT
   set count_of_criticals = my_counter;
end;
go
```

NOTES:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- Because the SQL procedure construct has no specific tab for comments, a full description of the procedure should be placed inline within the procedure body. Use the same format as for triggers including the type of object (ie. TRIGGER, PROCEDURE, etc.), the name of the procedure (ie. `count_up_criticals`), the last modified by username and date and a full description of what the procedure is for and how it does it.
- Include in the procedure description an example of its use (ie. *Usage*) so it is clear how this procedure should be used.

Note: See the section entitled: *CREATE PROCEDURE (SQL procedures)* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more information on SQL procedures.

External procedures

As mentioned above, external procedures are used to execute an external action on the host system — for example, an executable script — via the Netcool Process Agent.

EXAMPLE:

On insert of a certain type of event (`Class = 9900`), *Widgetcom* business requirements stipulate that an inventory script must be executed to carry out a number of inventory actions in response to the receipt of the event. The inventory script requires the hostname (`Node`) of the event and the asset number (`AssetNum`) to be passed to it as arguments.

The Netcool Administrator elects to create an external procedure to launch the inventory script and a pre-insert database trigger that will detect the insertion of this type of event and then execute the procedure.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
------------------------------------------------------------------
-- CREATE PROCEDURE TO LAUNCH THE INVENTORY SCRIPT
--
-- EXTERNAL PROCEDURE run_inventory_script
-- Created by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
-- This external procedure takes two arguments - a hostname and an
-- asset number - and then passes these to an external script named
-- inventory.sh.  This procedure is called by the pre-insert trigger
-- run_inventory_for_class_9900 for all events where Class = 9900.
--
CREATE OR REPLACE PROCEDURE run_inventory_script (
   IN node_name CHAR(64),
   IN asset_number INTEGER )
EXECUTABLE '$OMNIHOME/utils/inventory.sh'
HOST 'localhost'
USER 1000
GROUP 0
ARGUMENTS node_name, asset_number;
go
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
--------------------------------------------------------------------
-- CREATE TRIGGER TO LAUNCH INVENTORY SCRIPT
CREATE OR REPLACE TRIGGER run_inventory_for_class_9900
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER run_inventory_for_class_9900
Last modified by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This trigger executes the procedure run_inventory_script on the
local host for each Class 9900 event newly inserted into the
ObjectServer.'
BEFORE INSERT ON alerts.status
FOR EACH ROW
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
begin

   -- CHECK CLASS OF INCOMING ROW
   if (new.Class = 9900) then

         -- EXECUTE INVENTORY SCRIPT
         execute run_inventory_script (
               new.Node,
               new.AssetNum);
   end if;
end;
go
```

NOTES:

- In order for external procedures to execute correctly, the ObjectServer must be running under Process Agent control with the properties `PA.Name`, `PA.Username` and `PA.Password` set. (See the section entitled *ObjectServer properties* for more details on these properties.)
- The external procedure creation construct does not make provision for a comment section. As a result, a full description including last modified date and username cannot be created within the procedure itself. This information however can still be included within the solution delivery SQL file used to build the ObjectServer as per the above example, and fully documented in the solution detailed design document (DDD).
- This solution uses a pre-insert database trigger to check the event prior to insertion and launches a shell script if it is of Class 9900. This solution may adversely affect the ObjectServer insert rate as well as the physical machine loading if a large number of Class 9900 events are received in a short period of time. Solutions such as these should be created with care therefore.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: See the section entitled *External procedures* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more information about external procedures.

Default procedures

There are a number of default procedures present in the ObjectServer. This section outlines their purpose and provides some guidelines for their use.

automation_enable, automation_disable procedures

These procedures are for the enabling and disabling of the `primary_only` trigger group on a backup ObjectServer that participates in an Aggregation layer failover pair. These procedures are called from the three signal triggers

- `backup_counterpart_down`
- `backup_counterpart_up`
- `backup_startup`

The first two signal triggers are raised on a backup Aggregation ObjectServer when the failover bidirectional Gateway detects the counterpart (ie. the primary Aggregation ObjectServer) is down or up respectively.

If the primary Aggregation ObjectServer goes down, the procedure `automation_enable` is called — which, in turn, *enables* the `primary_only` trigger group. If the primary Aggregation ObjectServer comes back up, the procedure `automation_disable` is called — which, in turn, *disables* the `primary_only` trigger group.

The third signal trigger above: `backup_startup` is set to fire when the default signal startup is raised when the ObjectServer starts up. This in turn calls the procedure `automation_disable`, which, in turn, disables the `primary_only` trigger group. This ensures that the `primary_only` trigger group is always disabled initially when the backup ObjectServer starts up.

NOTES:

- These three signal triggers should *only* be enabled on a backup Aggregation ObjectServer.
- These three signal triggers are all disabled by default when an ObjectServer is initially created. When the multitier configuration SQL file `aggregation.sql` is applied, it detects whether or not the current ObjectServer being built or configured is a backup ObjectServer. It does this by testing to see if its name ends with the string "_B". If it does, these three triggers are enabled. If it does not, these triggers are left disabled.

jinsert procedure

This procedure exists to provide a mechanism by which journal entries can be created and associated with an event from a tool, `nco_sql` command prompt, trigger, or procedure.

To call this procedure, the event's Serial, a valid user ID, a timestamp and the journal entry must be passed as arguments, like in the following example:

```
execute jinsert(
   old.Serial,
   %user.user_id,
   getdate(),
   'This is my journal entry');
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: Journals must be used sparingly as overuse can significantly impact ObjectServer performance. After installing any functionality that creates journal entries (particularly automated ones like triggers), the system profiling should be carefully monitored for a while to ensure no significant performance degradation occurs. Periodic checking of the `alerts.journal` table should be undertaken as part of normal health checks in any case. Also see the section entitled *ObjectServer housekeeping* later in this chapter for guidance relating to maintaining healthy levels of table data.

Indexes

Indexes (or indices) are used to shorten the time required to perform searches or scans on ObjectServer tables by providing faster access to fields that have indexes created against them.

When an index is created, the ObjectServer builds an ordered data structure that contains direct links to the respective rows in the table the index is referencing. When that field is accessed, the ordered data structure is parsed over to access the row directly, instead of performing a methodical scan of the entire table.

The construction and maintenance of the data structure does incur some overhead in terms of performance, however the savings gained by not having to scan entire tables usually outweighs this overhead. The performance gains also increase the more the table size grows.

Indexes should not be simply created for every field. The guidance is to only index one or two fields in each table at most, that commonly appear in the WHERE clauses of queries.

Note: The primary key fields of all tables are indexed automatically by the ObjectServer on creation and so do not need to be indexed manually. Some default tables have additional important fields indexed also — for example: the `Serial` field in the `alerts.status` table.

When to use indexing

Indexing should be considered when one or more of the following apply:

- A non-indexed field is used extensively within ObjectServer triggers, tools, or procedures.
- A table being scanned on a non-indexed field by an ObjectServer trigger, tool or procedure is large, or is likely to grow to a large size.
- A necessary, frequent SQL operation will carry out a scan/search on a non-indexed field in a large table.

How to determine what effect indexing has made

The ObjectServer's trigger statistics and profiling logs should be reviewed to determine what effect adding an index has made, before and after the index has been created. A positive effect is indicated by a reduction in the overall ObjectServer time used.

EXAMPLE:

After a routine review of the ObjectServer trigger statistics logs, the Netcool Administrator at *Widgetcom* notices that a recently added custom trigger is consuming a disproportionate amount of the ObjectServer's time:

```
Tue Sep 06 14:07:19 2011: Trigger Group 'widgetcom_triggers'
Tue Sep 06 14:07:19 2011: Trigger time for 'scan_assets': 20.412565s
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

After an analysis of the pre-insert database trigger `scan_assets`, the Netcool Administrator identifies that the `AssetNum` field is being used by the trigger to locate and count the number of events in the `alerts.status` table that have the same asset number as the event that is about to be inserted.

```
-- COUNT UP NUMBER OF ROWS WITH SAME ASSET NUMBER AS INCOMING ROW
for each row same_asset_num in alerts.status where
      same_asset_num.AssetNum = new.AssetNum
begin
      -- INCREMENT THE COUNTER
      set num_asset_rows = num_asset_rows + 1;
end;
```

In addition to this, the trigger performs a further scan of the `alerts.status` table to calculate how many events fall into the same location ID range.

```
-- COUNT UP NUMBER OF ROWS IN THE SAME ID RANGE AS INCOMING ROW
for each row same_loc_range in alerts.status where
      same_loc_range.LocationID >= min_value and
      same_loc_range.LocationID <= max_value
begin
      --INCREMENT THE COUNTER
      set num_loc_rows = num_loc_rows + 1;
end;
```

Since the first query is performing a lookup based on equality, the Netcool Administrator decides to create a HASH index on the field `AssetNum`. Since the second query type is performing a lookup based on a range, the Netcool Administrator decides to create a TREE index on the field `LocationID`.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-------------------------------------------------------------------
-- CREATE INDEXES TO ASSIST EXECUTION OF TRIGGER scan_assets
--
-- INDEXES AssetNumIdx, LocationIDIdx
-- Created by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
-- The trigger scan_assets performs an equality type scan of
-- alerts.status on the AssetNum field - hence a hash index has been
-- created on this field to aid in trigger execution.
-- The trigger scan_assets performs a range type scan of
-- alerts.status on the LocationID field - hence a tree index has
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
-- been created on this field to aid in trigger execution.
--
CREATE INDEX AssetNumIdx ON alerts.status USING HASH (AssetNum);
go

CREATE INDEX LocationIDIdx ON alerts.status USING TREE (LocationID);
go
```

After carrying out the necessary due diligence of testing the new indexes in the test system, they are deployed into production. On reviewing the ObjectServer trigger statistics logs for the same time on the same day of the week, the Netcool Administrator notices a dramatic drop in the amount of time consumed by the trigger:

```
Tue Sep 13 14:07:19 2011: Trigger Group 'widgetcom_triggers'
Tue Sep 13 14:07:19 2011: Trigger time for 'scan_assets': 3.412565s
```

NOTES:

- The index creation construct does not make provision for a comment section. As a result, a full description including last modified date and username cannot be created within the index itself. This information however can still be included within the solution delivery SQL file used to build the ObjectServer as per the above example, and fully documented in the solution detailed design document (DDD).
- A standard naming convention should be followed when creating index names. The one used in this example follows the *IBM Netcool/OMNIbus 8.1 Administration Guide*, that is, appending "Idx" to the end of the field name.

Note: See the section entitled: *Creating indexes* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more detailed information about indexes, including when each of the two different types of index should be used and restrictions on index creation.

Also see the sections entitled: *Review and amend your SQL queries, and create a selection of well-designed, efficient indexes* and *Indexing guidelines* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for more information about identifying candidate fields for indexing and some usage examples of indexes respectively.

Tools, menus, and prompts

This section contains some guidelines for the creation of ObjectServer tools, menus, and prompts. Note that most of these guidelines apply to Native Event List tools only. Some concepts relating to SQL tools however apply to both Native Event List and IBM Netcool/OMNIbus WebGUI tools.

The following list contains some tips for tool, menu, and prompt creation:

- To ensure that Native Event List views are updated after the execution of an SQL tool, always include the following SQL command as the last line:

  ```
  flush iduc;
  ```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This causes any Native Event List running the tool to refresh itself with the ObjectServer immediately — ensuring the user's view is updated.

- It is recommended to build Native Event List tools and menu items via the IBM Netcool/OMNIbus Administrator utility.
- The recommended method for copying tools and menu items from one ObjectServer or environment to another is via the `nco_confpack` utility. This can be found in the `$OMNIHOME/bin/` directory. Instructions for its use can be found in the *IBM Netcool/OMNIbus 8.1 Administration Guide*, or by running the utility with the `-help` switch.
- When creating a Native Event List tool, ensure that the *Access* tab is configured to restrict access to those users or groups that required it. The default is to allow everyone access to the tool.
- When creating an external action Native Event List tool, ensure that the *Platform* tab is configured correctly so that the tool cannot inadvertently be run on the wrong platform.
- When creating a Native Event List tool, ensure that the *Description* tab is fully completed in the same way as for triggers. It should contain at a minimum the tool name, date of creation or last modified date, the user who created or last modified the tool and a full description of what the tool is for and what it does. The tool description should also include information about any menus or prompts this tool uses.

  An example description follows:

```
TOOL AssignLocation
Created by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
This tool is intended for use by operations administrators and
allows an event or group of selected events to be assigned to a
regional location.  It makes use of the fixed choice prompt
AssignLocationPrompt for this purpose.  This tool is a context
sensitive tool and appears in the Alerts menu with the label
Assign Location.
```

- The menu and prompt creation constructs do not make provision for a comment section. As a result, a full description including last modified date and username cannot be created within the menu or prompt itself. This information however should be fully documented in the solution detailed design document (DDD).
- The solution detailed design document (DDD) should contain detailed information about every custom tool, menu and prompt created in the solution.

Adding custom fields

This section contains some guidance relating to the adding of custom fields to an IBM Netcool/OMNIbus installation.

Include custom fields in solution delivery SQL file

Custom fields should be added to the solution delivery SQL file, ideally at or near the top, so that they are automatically created when an ObjectServer is built or rebuilt. Solution delivery SQL file entries should contain adequate descriptions for each new field as per the following example:

```
---------------------------------------------------------------
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
-- CREATE CUSTOM FIELDS IN AGGREGATION OBJECTSERVER
--
-- Created by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
-- AssetNum     This field stores the Widgetcom asset number of
--              device the event has originated from.
-- LocationID   This field stores the unique Widgetcom location
--              identification code.  Note that this field has
--              conversions associated with it for the benefit of
--              Event List users (see conversions below).
-- LocationDesc This field stores a description of the Widgetcom
--              site this device is located at.  It is populated
--              by IBM Netcool Impact after a lookup in the
--              IBM Maximo asset database.
--
ALTER TABLE alerts.status
   ADD COLUMN AssetNum INTEGER
   ADD COLUMN LocationID INTEGER
   ADD COLUMN LocationDesc VARCHAR(40);
go
```

Include custom field conversions in solution delivery SQL file

Conversions need only be added to the Aggregation ObjectServers, since they are automatically propagated to Display ObjectServers via the Aggregation to Display ObjectServer Gateways. They may also be added to the Collection layer ObjectServers for the sake of Netcool Administrators who may occasionally connect a Native Event List to the Collection layer for debugging purposes, for example.

Conversions should be included in the solution delivery SQL file underneath the field additions section. The following example follows on from the previous example:

```
---------------------------------------------------------------
-- CREATE CUSTOM FIELD CONVERSIONS FOR FIELD LocationID
--
-- Created by: J Smith (jsmith) - Netcool Admin - 5-Aug-2011
-- Possible values: 0 - London (default)
--                  1 - New York
--                  2 - Auckland
--
INSERT INTO alerts.conversions (
   KeyField,
   Colname,
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
    Value,
    Conversion)
values (
    'LocationID0',
    'LocationID',
    0,
    'London');
go

INSERT INTO alerts.conversions (
    KeyField,
    Colname,
    Value,
    Conversion)
values (
    'LocationID1',
    'LocationID',
    1,
    'New York');
go

INSERT INTO alerts.conversions (
    KeyField,
    Colname,
    Value,
    Conversion)
values (
    'LocationID2',
    'LocationID',
    2,
    'Auckland');
go
```

Add custom fields to Gateway mappings

One of the steps of adding custom fields to the IBM Netcool/OMNIbus installation is to ensure that the new fields also get added to the appropriate Gateway mapping files so that the fields' contents get propagated around the IBM Netcool/OMNIbus environment.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

If you have a Collection layer in place, and Probes are populating the fields connected to the Collection layer, then you will need to add the new field mappings to the Collection to Aggregation Gateway mapping file.

Once at the Aggregation layer, the new fields will also have to be added to the bidirectional failover Gateway mapping file also to ensure that the primary and backup Aggregation ObjectServers remain synchronised.

Finally, if you are using a Display layer and these are fields that the end users are going to need to be able to see, you will also have to add them to the Aggregation to Display Gateway mapping file.

Note that in all standard multitier configuration Gateway mapping files, there is a designated area for adding custom field mappings to the `alerts.status` table. For example:

```
#################################################################
#
#  CUSTOM alerts.status FIELD MAPPINGS GO HERE
#
#################################################################

   'AssetNum'         = '@AssetNum' ON INSERT ONLY,
   'LocationID'       = '@LocationID',
   'LocationDesc'     = '@LocationDesc',

#################################################################
```

If the data contained in the field will never be updated — for example, in this case a device's asset number is fixed — and the value has been set prior to the event being inserted (in the Probe rules for example), then it is prudent for performance reasons to append ON INSERT ONLY to the field mapping as in the above example. This will prevent the Gateway unnecessarily sending these fields over on every update.

Care must be taken however when carrying out event enrichment after insert with IBM Netcool/Impact, for example. If the failover bidirectional Aggregation ObjectServer Gateway forwards the event over to the backup Aggregation ObjectServer *before* the event has been enriched, and that field is set to ON INSERT ONLY, the enriched information will not propagate over. In such cases, it is advisable to leave ON INSERT ONLY off these field mappings to avoid unwanted race conditions.

The term *race condition* refers to the scenario where the action of one process — for example: the propagation of the field data from the primary Aggregation ObjectServer to the backup — is affected by action of one or more other processes — for example: the enrichment of the field by IBM Netcool/Impact, and vice versa. The resulting outcome will depend on which action happens to occur first. The word “race” is used in this expression to describe the apparent “racing” of these two processes against each other.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: For Collection to Aggregation and Aggregation to Display Gateway `alerts.status` table mappings, putting ON INSERT ONLY after field mappings has no effect. This is because inter-tier Gateways convert all updates to inserts anyway due to the following option included in the respective table replication files:

```
SET UPDTOINS CHECK TO FORCED
```

See the section entitled *Gateways* later in this document for more guidance on configuring Gateway mappings and table replication files.

Create generic fields for future use

When custom fields are added to an IBM Netcool/OMNIbus installation, they must also be added to the interconnecting Gateway mappings as described above. For these additions to take effect, Gateways must be restarted — which incurs resynchronisation load and hence interruptions to production.

Also, the modification to production system configuration usually requires a maintenance window, particularly where components need restarting. Making changes to production systems carries risk — for example: if there is an error introduced into the configuration by the changes made; or if the configuration places unexpected loading on the system for whatever reason.

One way to reduce the occurrence of configuration changes and to limit the impact of installing changes in production when additional fields are needed is to pre-add a number of generic integer and varchar fields to the configuration for future use. Both field types incur tiny cost to the ObjectServer memory footprint if they are unused and, if pre-added to the ObjectServer and Gateway configuration before needed, mean that they are installed and ready for use immediately when needed, without the need to restart any components.

When these fields are needed, the way they appear in an Event List can be modified by simply adding column visuals to the Aggregation ObjectServer, which will then propagate automatically to the Display layer ObjectServers via the Aggregation-to-Display ObjectServer Gateways. The addition of column visuals can be done on-the-fly and doesn't require the restart of any components.

EXAMPLE:

Widgetcom are planning to put into production a new IBM Netcool/Impact policy that will transition events through an event state model. Each event individually needs to store its current state in the event model. Rather than creating a new field to hold this state information, the Netcool Administrator elects to use one of the generic integer fields `Int01` instead for this purpose. Since the state values consist of a pre-defined set of values, an integer field is selected for this task rather than a string field, since integer fields are generally more efficient to transfer and use.

To ensure that the field appears meaningful in the context of the Event List, the following column visual is also added to the *Widgetcom* solution delivery SQL file:

```
---------------------------------------------------------------
-- CREATE FIELD COLUMN VISUAL FOR WIDGETCOM STATE MODEL FIELD
--
-- Created by: J Smith (jsmith) - Netcool Admin - 12-Sep-2011
-- The generic field Int01 is being used for the Widgetcom event
-- state model and so a column visual is needed so that the column
-- in the Event List shows a more meaningful name.  The field
-- will be labelled State, have a default width of 10, a maximum
-- width of 40 and will be left-justified for both the column
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
-- heading and the column contents.
insert into alerts.col_visuals (
   Colname,
   Title,
   DefWidth,
   MaxWidth,
   TitleJustify,
   DataJustify)
values (
   'Int01',
   'State',
   10,
   40,
   1,
   1);
go
```

NOTES:

- The justification codes for the field values `TitleJustify` and `DataJustify` are: left, centre and right for 0, 1 and 2 respectively.
- In this example, conversions should also be created for the `Int01` field, just as was done in a previous example, so that the field shows a more meaningful textual state of the events in the Event List (eg. 'UNPROCESSED', 'OPEN', 'CLOSED') rather than merely the integer values (eg. 0, 1, 2).
- Although comments are provided in-line in the solution delivery SQL file documenting how these fields are being used, appropriate notes should also be added to the solution detailed design document (DDD) detailing how they're being used and why.

Note: The creation of generic fields for future use is entirely optional. It is included in this section merely as an optional technique to use if desired.

User and group creation

All users and groups should be created in the primary Aggregation ObjectServer, or the backup Aggregation ObjectServer if the primary is down. Once created, the newly created user or group will automatically propagate to the counterpart Aggregation ObjectServer and Display ObjectServers, when present. This process ensures that the users and groups are created with the same UID or GID respectively.

Normal users are not supposed to connect to the Collection layer, only administrative or “System” users. In the latter case, administrative users must first be set up before they can connect. Since the Collection to Aggregation Gateways only propagate data from Collection to Aggregation, simply creating the user on the Aggregation layer will not result in the user being created at the Collection layer. Hence, in a system where a Collection layer is present, any administrative users also need to be created manually on each Collection ObjectServer.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Process for user and group creation

To ensure that UIDs and GIDs are consistent across all ObjectServers for any given user or group, a process for user and group creation needs to be followed to ensure this is so.

Consistency of UIDs and GIDs is especially important in a multiple IBM Netcool/OMNIbus partition environment, where multiple 1, 2 or 3-tiered ObjectServer installations are viewed from a common IBM Netcool/OMNIbus WebGUI server.

Note: For more information on multiple partition IBM Netcool/OMNIbus systems for large scale or geographically distributed scenarios, see the Best Practices document available on the IBM Netcool/OMNIbus developerWorks wiki site: http://ibm.biz/nco_bps

It is also important to ensure consistency of UIDs and GIDs when viewing historic archive event data since user actions are referenced by UID and group assignment is referenced by GID in the historical event data.

Users and groups can be created manually on each ObjectServer via the IBM Netcool/OMNIbus Administrator tool. If done this way, it is possible to specify the UID and password for user creation — and the GID for group creation, and ensure uniformity across the deployment. This method of user and group creation can be time consuming for an administrator however and prone to errors being made, especially if there are many Collection ObjectServers present.

An alternative, recommended process for ObjectServer user and group creation involves employing a user creation SQL file and then running it against the target ObjectServers via the command line SQL interface. This will ensure uniformity of UIDs and GIDs across the system.

This is achieved by creating an SQL file that contains the SQL necessary to create the users or groups. The SQL language allows for the UIDs or GIDs to be specified at the point of creation or users or groups respectively.

The user creation SQL file should be run against all the following within an IBM Netcool/OMNIbus deployment:

- Primary Aggregation ObjectServer in each IBM Netcool/OMNIbus partition that exists;
- Every Collection ObjectServer (both primaries and backups) for administrative users that may access the Collection layer.

Note: It is recommended to encrypt the password before creating the user creation SQL file entry using `nco_g_crypt`. Access to the user creation SQL file should also be restricted.

EXAMPLE:

The Netcool Administrator at *Widgetcom* needs to manually create a new user group called “SysOps” for the System Operations team and assign a new user George Tamariki to the group. The new user “george” should be assigned an initial password of “g30rge”. Finally, George needs to be set up as an IBM Netcool System user and will need to access the Collection layer ObjectServers.

A user creation SQL file is created with the following contents and then is run against the primary Aggregation ObjectServer and all Collection layer ObjectServers:

```
CREATE USER 'george'
   ID 10001
   FULL NAME 'George Tamariki'
   PASSWORD 'DLBBFFBFBBEN' ENCRYPTED
   PAM FALSE;
go
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
CREATE GROUP 'SysOps'
   ID 1001
   COMMENT 'Systems Operations Team'
   MEMBERS 'george';
go

ALTER GROUP 'System'
   ASSIGN MEMBERS 'george';
go
```

Using VMMSYNC on WebGUI

IBM Netcool/OMNIbus WebGUI has a feature called VMMSYNC that allows the automatic creation of users on the ObjectServer infrastructure whenever WebGUI roles are assigned to users or groups referenced in the federated repository in use.

An example of this might be a corporate LDAP server: whenever a user is added to the "Netcool" group, the VMMSYNC process will create that user on the ObjectServer infrastructure so that when that user logs into WebGUI, opens an Active Event List and runs tools or adds journal entries, they will be successful. Without corresponding user creation on the ObjectServer layer, users would be able to log in, but would have limited capabilities. Similarly, whenever a user is removed from the "Netcool" group on the LDAP server, the VMMSYNC process will remove that user on the ObjectServer infrastructure.

Using VMMSYNC therefore is a very convenient method for automatically adding and removing ObjectServers users that may be managed via an external user management tool.

Since user and group creation by the VMMSYNC process does not specify UID or GID to use, the ObjectServer defaults to using the "next available" UID or GID. This will be the lowest numerical UID or GID that is not already being used.

MULTIPLE DATASOURCES

In a single datasource scenario where there is just a single 1, 2 or 3-tiered IBM Netcool/OMNIbus system being accessed by IBM Netcool/OMNIbus WebGUI, there is no potential for problems.

When there are multiple datasources — also known as IBM Netcool/OMNIbus partitions — care must be taken to ensure that users and groups are created consistently by the VMMSYNC process to ensure alignment of UIDs and GIDs to users and groups respectively across the datasources/partitions.

This can be achieved by adhering to the following rules:

- All *manually* created users — that is, users not created via the VMMSYNC process — should be created with a UID higher than 10,000. This will ensure the first 10,000 UIDs are available for the exclusive use of VMMSYNC.
- All *manually* created groups — that is, groups not created via the VMMSYNC process — should be created with a GID higher than 1,000. This will ensure the first 1,000 GIDs are available for the exclusive use of VMMSYNC.
- All datasources/partitions should be online when the user or group WebGUI role is assigned on the IBM Netcool/OMNIbus WebGUI server running VMMSYNC. This is necessary to ensure the user or group will be created successfully on all datasources/partitions.
- All datasources/partitions should be checked after the role assignment has been granted to ensure that UIDs and GIDs are aligned across the datasources. If there is a mismatch, the users/groups

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

should be deleted of the ObjectServer sub-systems and the VMMSYNC process re-run to re-create the affected users or groups.

Note: Where it is anticipated that the number of users will exceed 10,000 or the number of groups will exceed 1,000, the above starting values should be increased accordingly.

EXAMPLE:

Following on from the previous example, the *Widgetcom* Netcool Administrator has an IBM Netcool deployment that contains three IBM Netcool/OMNIbus partitions with a Tivoli Integrated Portal server running IBM Netcool/OMNIbus WebGUI 8.1 with VMMSYNC enabled.

The user creation SQL file created in the previous example is run against the primary Aggregation ObjectServer in each partition and all Collection layer ObjectServers in all partitions. The new user “george” has a UID of 10001 and the new group “SysOps” has a GID of 1001 so that the assigned values do not interfere with the VMMSYNC user and group creation process.

IDUC port for firewall configuration

When an ObjectServer starts, it checks the interfaces file to see which port it should run on (eg. 4100). After start-up, the ObjectServer then randomly selects a second port from the ephemeral port range through which to publish IDUC data (eg. 32123).

When an IDUC client, such as a Gateway reader or a Native Event List, connects to the ObjectServer on its main port as per the interfaces file, it will query the ObjectServer for the port number of its IDUC port. When the ObjectServer responds, the client will open a second connection to the IDUC port and listen out for IDUC notifications.

Before it is possible to connect an IDUC client to an ObjectServer through a firewall, the firewall must be configured to allow the connections. Because the IDUC port is randomly selected however, it is not possible to set up a firewall rule that will allow connection on that port since it is unknown beforehand.

The solution is to set the property `Iduc.ListeningPort` in the ObjectServer to a fixed value, which will cause the ObjectServer to only use this port number for its IDUC port, and hence make the specification of a firewall rule possible.

Note: Also see the section entitled *ObjectServer properties* in *Chapter 4* of this document for more information on ObjectServer properties including the property mentioned here: `Iduc.ListeningPort`.

ObjectServer HTTP and OSLC ports

The ObjectServer has the option to open an HTTP (or HTTPS) port or an OSLC port through which SQL commands can be executed by a third-party client.

It is important to remember that care should be taken when using these features, especially via automated processes, that the load incurred by the SQL commands does not overload the ObjectServer.

Note: See the *IBM Netcool/OMNIbus 8.1 Administration Guide* and the *IBM Netcool/OMNIbus ObjectServer OSLC Interface Reference Guide* for more information about how to configure and use the ObjectServer HTTP and OSLC ports.

ObjectServer housekeeping

One of the most significant factors with regards to ObjectServer performance is the number of unique events in the `alerts.status` table. As such, it is essential that measures are put in place to ensure event numbers never grow to a point where the ObjectServer is overloaded.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: This principle applies to all tables in the ObjectServer, particularly custom tables, the `alerts.details` table, and the `alerts.journal` table. The housekeeping automations included in this section are now shipped with IBM Netcool/OMNIbus 8.1 in the extensions directory.

ExpireTime

The `ExpireTime` field is a default field used in conjunction with the default `expire` trigger that provides an automated mechanism for the expiration of events. The `expire` trigger clears the severity (ie. sets it to zero) of any event whose last occurrence (ie. `LastOccurrence`) was more than `ExpireTime` number of seconds ago.

The values that are assigned to events will vary from one business to another and may be specified based on event type, event severity, some other criteria, or a combination. The key point here is that it is a best practice that the `ExpireTime` field should be set for *every* event in the ObjectServer. No events should persist indefinitely in the ObjectServer.

This concept is a best practice for event expiry.

It is a best practice to implement functional logic as early in the system as possible — for example: Probe rules files initially. In some cases, however, there may be event sources in which the `ExpireTime` field is not set. As a *"catch-all"* mechanism, it is recommended to also have a trigger present that will set the `ExpireTime` field for those events where it is not set.

EXAMPLE:

Widgetcom have drafted a new event expiry policy: Critical events are to be retained up to a week, Major events up to 5 days, Minor events up to 3 days, Warning events 1 day and Indeterminate events 4 hours. These periods should be based on each event's last occurrence.

The Netcool Administrator decides to implement this event expiry policy firstly in the Probe rules, to ensure that all events coming from those Probes have the `ExpireTime` field set, and secondly in a *"catch-all"* temporal trigger, that sets the `ExpireTime` field for events where it is not set.

The following Probe rules file code fragment is added to a new file `expiry_policy.rules` that will be included by each of the Probe's rules files. This means that if a change to the event expiry policy is required, the change only needs to be made in one place.

```
####################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 13-Sep-2011
# This section of Probe rules file implements the event expiry
# policy for Widgetcom.
# Widgetcom event expiry policy as at 13-Sep-2011:
#     Critical events - 7 days
#     Major events - 5 days
#     Minor events - 3 days
#     Warning events - 1 day
#     Indeterminate events - 4 hours
####################################################################
# SET ExpireTime FOR EVENTS THAT HAVE A NON-ZERO SEVERITY AND WHERE
# ExpireTime IS NOT ALREADY SET OR SET TO ZERO
if (int(@Severity) > 0 &&
   (match(@ExpireTime, "") || match(@ExpireTime, "0")))
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
{
   switch(@Severity) {
         case "5":
               @ExpireTime = 604800
         case "4":
               @ExpireTime = 432000
         case "3":
               @ExpireTime = 259200
         case "2":
               @ExpireTime = 86400
         case "1":
               @ExpireTime = 14400
         default:
               # THERE SHOULD NOT BE ANY OTHER SEVERITY EVENTS
               log(ERROR, "Event with non-standard Severity (" +
                     @Severity +
                     ") detected, Identifier = " +
                     @Identifier)
   }
}
```

The above file is included by each of the Probes by adding the following to the end of each of the Probes' rules files:

```
include "$OMNIHOME/etc/probes/expiry_policy.rules"
```

Note: See the section entitled *Probe configuration file locations* later in this document for more information on the location of custom Probe configuration file locations.

The trigger does not need to execute very regularly since the shortest expiry time is only 4 hours, so the nearest prime value to 10 minutes is selected (ie. 599 seconds). The default trigger expire will then remove events once they have reached their respective expiration times.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
------------------------------------------------------------------
-- CREATE THE WIDGETCOM DEFAULT EXPIRATION TRIGGER
CREATE OR REPLACE TRIGGER set_expiretime
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER set_expiretime
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
Last modified by: J Smith (jsmith) - Netcool Admin - 13-Sep-2011
This trigger sets the ExpireTime field for all events where it is
not yet set.  It works in conjunction with the default expire
trigger to provide an automated event expiry mechanism.
Widgetcom event expiry policy as at 13-Sep-2011:
     Critical events - 7 days
     Major events - 5 days
     Minor events - 3 days
     Warning events - 1 day
     Indeterminate events - 4 hours
     Clear events are to be ignored by this trigger.'
EVERY 599 SECONDS
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
begin

   -- FIND ROWS WHERE ExpireTime IS NOT YET SET AND SET ExpireTime
   -- BASED ON EVENT SEVERITY - IGNORE CLEARED EVENTS
   for each row unexpired in alerts.status where
         unexpired.ExpireTime = 0 and
         unexpired.Severity != 0
   begin

         -- CRITICAL EVENTS: 7 DAYS (604800 SECONDS)
         if (unexpired.Severity = 5) then

               set unexpired.ExpireTime = 604800;

         -- MAJOR EVENTS: 5 DAYS (432000 SECONDS)
         elseif (unexpired.Severity = 4) then

               set unexpired.ExpireTime = 432000;

         -- MINOR EVENTS: 3 DAYS (259200 SECONDS)
         elseif (unexpired.Severity = 3) then

               set unexpired.ExpireTime = 259200;

         -- WARNING EVENTS: 1 DAY (86400 SECONDS)
         elseif (unexpired.Severity = 2) then
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
                set unexpired.ExpireTime = 86400;

          -- INDETERMINATE EVENTS: 4 HOURS (14400 SECONDS)
          elseif (unexpired.Severity = 1) then

                set unexpired.ExpireTime = 14400;
          end if;
    end;
end;
go
```

Monitoring row numbers

In addition to ensuring that all events expire, ObjectServer tables should also be monitored in terms of the number of rows contained in each to ensure none grow beyond their maximum handling thresholds. Key tables include: the main event table (`alerts.status`), the event details table (`alerts.details`), the event journal table (`alerts.journal`) and any custom tables created.

With respect to the `alerts.status` table, a "small" Netcool system will have up to 10,000 standing rows at any time, a "medium" Netcool system will have between 10,000 and 50,000 and a "large" Netcool system anything more than that.

The maximum number of events a system will be able to handle will depend on a variety of factors including: the platform being used, the CPU speed, the load of any custom triggers, the load of any connected Probes and Gateways and the load of any connected applications such as IBM Netcool/Impact, IBM Tivoli Network Manager IP Edition or IBM Tivoli Business Service Manager (TBSM). Another significant factor is the number of rows in any other tables (eg. custom tables) — and how those tables are used by both connecting clients and internal processes such as triggers.

Because there are so many variables that affect ObjectServer performance, benchmark testing should be carried out in the target environment (ie. platform, machine specifications, configuration customisations such as triggers) to establish what maximum row limits are reached before ObjectServer performance is impacted. These numbers can then form a basis for defining row count thresholds in the key tables before automated action is taken, such as automated event and/or data deletion.

The automated deletion of event data is a drastic measure to take, however one must consider the alternative: the event management system going down or being unusable due to an event storm. In this situation, data deletion may be drastic, but may be *necessary* as a last-resort self-protection mechanism.

Based on the maximum table numbers, thresholds should be agreed to by the business to establish at what point any automated function will begin the process of purging expendable data. Also, the business management must agree on what data is considered "expendable" under such circumstances. Thresholds should be set at a suggested 80% of maximum values to allow the system time to begin reacting before the point at which it is overwhelmed.

EXAMPLE:

Widgetcom has carried out benchmark testing and has found it has the following maximum limits on the tables indicated before ObjectServer performance begins to be affected:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServer table	Maximum threshold	80% of maximum threshold
alerts.status	80,000	64,000
alerts.journal	150,000	120,000
alerts.details	100,000	80,000

Based on the above figures, the Netcool Administrator gains approval from business management to implement an automated process that will purge rows from the ObjectServer on a per-table basis, if that table's 80%-below-maximum-threshold is reached. It is agreed that the automated action(s) for each table will be as follows:

ObjectServer table	Action(s) if table 80%-below-threshold is breached
alerts.status	• Any event below Severity 3 will be deleted (excluding Type = 2 resolutions) • Any event older than 5 days will be deleted
alerts.journal	• Any journal older than 5 days will be deleted
alerts.details	• All details will be deleted

In order to provide a self-protection mechanism, the Netcool Administrator creates the following temporal trigger that will carry out the associated actions as per the event threshold breach policy. The trigger is set to run every 10 minutes at the nearest unused prime number value (ie. 601 seconds).

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-----------------------------------------------------------------
-- CREATE LOG FILE HANDLE FOR LOGGING EVENT PURGE OPERATIONS
CREATE OR REPLACE FILE deletion_log
'/opt/IBM/tivoli/netcool/omnibus/log/deletion_log.log'
MAXFILES 3
MAXSIZE 1 MBYTES;
go

-------------------------------------------------------------------
-- CREATE THE WIDGETCOM EVENT THRESHOLD PURGE TRIGGER
CREATE OR REPLACE TRIGGER purge_exceeded_events
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER purge_exceeded_events
Last modified by: J Smith (jsmith) - Netcool Admin - 13-Sep-2011
This trigger periodically checks the size of four key tables and
deletes data from those tables should it detect that a row threshold
has been breached.  Detection and deletion rules are on a per-table
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
basis.
Widgetcom event threshold breach policy as at 13-Sep-2011:
     alerts.status - >64,000
          delete Severity < 3 (excluding Type = 2 resolutions)
          delete events older than 5 days
     alerts.journal - >120,000
          delete journals older than 5 days
     alerts.details - >80,000
          delete all details'
EVERY 601 SECONDS
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
declare
  counter integer;
  now utc;
begin

  -- INITIALISE VARIABLES
  set counter = 0;
  set now = getdate();

  -- COUNT NUMBER OF alerts.status EVENTS
  for each row thisrow in alerts.status
  begin

        -- INCREMENT COUNTER
        set counter = counter + 1;
  end;

  -- CARRY OUT PURGE ON alerts.status IF THRESHOLD BREACHED
  if (counter > 64000) then

        -- DELETE EVENTS LOWER THAN Severity 3
        -- EXCLUDE Type 2 RESOLUTION EVENTS
        delete from alerts.status where
              Type != 2 and
              Severity < 3;

        -- DELETE EVENTS OLDER THAN 5 DAYS (432000 SECONDS)
        -- EXCLUDE TYPE 2 RESOLUTION EVENTS
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
        delete from alerts.status where
              Type != 2 and
              FirstOccurrence < (now - 432000);

        -- WRITE TO LOG THAT EVENTS WERE DELETED
        write into deletion_log(
              to_char(now) +
              ': event count is ' +
              to_char(counter) +
              ' - alerts.status table purge initiated.');
end if;

-- REINITIALISE VARIABLES
set counter = 0;

-- COUNT NUMBER OF alerts.journal ROWS
for each row thisrow in alerts.journal
begin

        -- INCREMENT COUNTER
        set counter = counter + 1;
end;

-- CARRY OUT PURGE ON alerts.journal IF THRESHOLD BREACHED
if (counter > 120000) then

        -- DELETE JOURNALS OLDER THAN 5 DAYS (432000 SECONDS)
        delete from alerts.journal where
              Chrono < (now - 432000);

        -- WRITE TO LOG THAT JOURNALS WERE DELETED
        write into deletion_log(
              to_char(now) +
              ': journal count is ' +
              to_char(counter) +
              ' - alerts.journal table purge initiated.');
end if;

-- REINITIALISE VARIABLES
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
    set counter = 0;

    -- COUNT NUMBER OF alerts.details ROWS
    for each row thisrow in alerts.details
    begin

          -- INCREMENT COUNTER
          set counter = counter + 1;
    end;

    -- CARRY OUT PURGE ON alerts.details IF THRESHOLD BREACHED
    if (counter > 80000) then

          -- DELETE ALL DETAILS
          delete from alerts.details;

          -- WRITE TO LOG THAT DETAILS WERE DELETED
          write into deletion_log(
                to_char(now) +
                ': details count is ' +
                to_char(counter) +
                ' - alerts.details table purge initiated.');
    end if;
end;
go
```

NOTES:

- It is recommended that significant automated actions are recorded in some way for auditing purposes. This example creates a deletion log file and writes to it each time it deletes any rows. The level of detail required in such audit logs will vary from business to business.
- The threshold deletion policy and date are included in the trigger description so that it is clear which version of policy is currently in place. The actions are further echoed in the code body also, so it is clear what actions are being carried out under what circumstances.

De-escalation of events

In the case of certain events, automated de-escalation should be considered so that events that are generated with an initially high severity can be automatically downgraded if they persist in the ObjectServer for long periods of time. Events that are candidates for this sort of functionality include warning or information type events.

EXAMPLE:

Widgetcom have decided that all events below severity 4 should gradually de-escalate until cleared based on the length of time in the ObjectServer. This will ensure that the lower severity events will

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

gradually lower in severity automatically (by 1 each time) until they are cleared and subsequently removed without any operator intervention. Type 2 resolution events should be ignored by this functionality.

The de-escalation policy is specified as follows:

Event Severity	**De-escalation rule**
3 (Minor)	LastOccurrence more than one day ago
2 (Warning)	LastOccurrence more than two days ago
1 (Indeterminate)	LastOccurrence more than three days ago

The Netcool Administrator creates the following temporal trigger that will carry out the associated actions as per the event de-escalation policy. The trigger is set to run every 10 minutes at the nearest unused prime number value (ie. 607 seconds).

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-------------------------------------------------------------------
-- CREATE THE WIDGETCOM EVENT DE-ESCALATION TRIGGER
CREATE OR REPLACE TRIGGER de_escalate_events
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER de_escalate_events
Last modified by: J Smith (jsmith) - Netcool Admin - 14-Sep-2011
This trigger periodically checks the time since the last occurrence
of events of severity 3, 2 and 1.  An event\'s severity is reduced
by 1 each time it is de-escalated.  Events of these severities will
gradually de-escalate over time until they are cleared.
Widgetcom event de-escalation policy as at 14-Sep-2011:
     Severity 3 - LastOccurrence > 1 day ago
     Severity 2 - LastOccurrence > 2 days ago
     Severity 1 - LastOccurrence > 3 days ago'
EVERY 607 SECONDS
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
declare
   now utc;

begin

   -- INITIALISE VARIABLES
   set now = getdate();
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
    -- PARSE OVER ALL SEVERITY 3, 2 and 1 EVENTS
    -- OLDER THAN ONE DAY
    -- EXCLUDE Type 2 RESOLUTION EVENTS
    for each row thisrow in alerts.status where
          thisrow.Type != 2 and
          thisrow.LastOccurrence < (now - 86400) and
          thisrow.Severity in (1, 2, 3)
    begin

          -- CHECK SEVERITY 3 EVENTS OLDER THAN 1 DAY (86400 SECS)
          if (  thisrow.Severity = 3 and
                thisrow.LastOccurrence < (now - 86400)) then

                -- DE-ESCALATE EVENT SEVERITY BY 1
                set thisrow.Severity = 2;

          -- CHECK SEVERITY 2 EVENTS OLDER THAN 2 DAYS (172800 SECS)
          elseif (    thisrow.Severity = 2 and
                      thisrow.LastOccurrence < (now - 172800)) then

                -- DE-ESCALATE EVENT SEVERITY BY 1
                set thisrow.Severity = 1;

          -- CHECK SEVERITY 1 EVENTS OLDER THAN 3 DAYS (259200 SECS)
          elseif (    thisrow.Severity = 1 and
                      thisrow.LastOccurrence < (now - 259200)) then

                -- DE-ESCALATE EVENT SEVERITY BY 1
                set thisrow.Severity = 0;
          end if;
    end;
end;
go
```

NOTES:

- This trigger is efficiently designed to parse over the `alerts.status` table only once. An IF-ELSEIF construct is contained within the FOR EACH ROW loop to test for the various conditions and the corresponding action carried out if they are satisfied. This is far more efficient than parsing over the `alerts.status` table three times (ie. have three FOR EACH ROW loops.)

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Backing up the ObjectServer

It is important to have a comprehensive back up régime in-place should the ObjectServer fail and need to be recovered - for example: after a hardware failure. It is recommended to have one or more of the following options in use to ensure a speedy recovery after an outage.

- Solution delivery SQL files - SQL files that can be used to build the ObjectServer
- SQL files exported via nco_osreport - can be used to rebuild the ObjectServer
- Backup copies of master_store.tab and table_store.tab files

This section covers the above options and provides a comprehensive strategy for backing up your ObjectServers and how to recover them in the event of an outage.

ObjectServer data files

When an ObjectServer shuts down, it writes its entire contents and state to the `master_store.tab` and `table_store.tab` files in the following directory:

`$OMNIHOME/db/`*`ObjectServer_name`*`/`

On restart, the ObjectServer process will read the contents of these two files into memory and resume its previous state. In addition to this process during start-up and shutdown, an ObjectServer will also "checkpoint" every 60 seconds where all persistent data is copied to checkpoint files (.chk). Between checkpoints, new and changed data is written to replay log files (.log). If the ObjectServer unexpectedly shuts down and is restarted, the ObjectServer process uses the combination of these files to rebuild the ObjectServer's state. On a clean shutdown of the ObjectServer, all changes and updates are incorporated into the `master_store.tab` and `table_store.tab` files prior to shutdown effectively capturing a complete and self-contained snapshot of the ObjectServer's state. After a clean shutdown, only the `master_store.tab` and `table_store.tab` files should be present in the ObjectServer's db directory.

Should the ObjectServer checkpoint files become corrupted, the corrupted checkpoint .chk and .log files can be deleted and the ObjectServer started based on the existing `master_store.tab` and `table_store.tab` files. This however may mean that the ObjectServer is restored to a very old state if the ObjectServer has not been restarted in a while.

About the automated backup function

To provide a contingency to this unlikely scenario, there exists in Netcool/OMNIbus an out-of-the-box feature that periodically backs up the ObjectServer to copies of the `master_store.tab` and `table_store.tab` files. The default setting for this feature is to back up the ObjectServer every 5 minutes and the function stores at most 2 copies of files that are taken 5 minutes apart.

Note: The number of previous stored backups is defined by the setting of the variable `num_backups` within the trigger `automatic_backup` and may be modified to suit requirements.

It is recommended to enable this feature for production environments, particularly at the Aggregation layer. The feature is enabled simply by enabling the trigger named: `automatic_backup`

Before enabling this feature, additional disk space should be provisioned to the disk partition where IBM Netcool/OMNIbus is installed to accommodate the additional backup files. The additional space required can be calculated by multiplying the existing `master_store.tab` and `table_store.tab` file sizes by 2 – for example: for the two additional sets of files. The automatic backup feature will automatically overwrite old backup files and only save the last two backups, hence no manual file management is required.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: Before the automatic backup system will successfully run, the Netcool administrator needs to create the backup directory: `$OMNIHOME/backup/`

The resulting `master_store.tab` and `table_store.tab` files are a fully self-contained snapshot of the ObjectServer at the moment the backup was taken. These files can be copied into the target ObjectServer's `db` directory prior to starting an ObjectServer to restore it to the desired state. These files can also be copied to another machine – for example: a test machine, provided the target hardware is of the same architecture.

About the nco_osreport utility

In addition to the automated backup function that backs up ObjectServer binary database files, there is another utility called `nco_osreport` that can be used to export an ObjectServer's configuration and contents to human-readable SQL files. These files can then be used to rebuild an ObjectServer, if needed, via the `nco_dbinit` utility.

The `nco_osreport` utility can be run as follows to capture an ObjectServer configuration:

```
nco_osreport -dbinit
      -norowdata alerts.status,alerts.details,alerts.journal
      -server AGG_P -user root -password netcool
```

Note: The `-norowdata` switch tells the utility not to export the row data from the status, details and journal table. Since the aim is to capture the configuration only, the table event data can be excluded to minimise the size of the output files.

The resulting SQL files can then be used in conjunction with the `nco_dbinit` utility to recreate the ObjectServer as per the following example:

```
$OMNIHOME/bin/nco_dbinit -server AGG_P
      -alertsdata alertsdata.sql
      -applicationfile application.sql
      -automationfile automation.sql
      -desktopfile desktop.sql
      -securityfile security.sql
      -systemfile system.sql
      -force
```

Note: The `-force` option instructs the `nco_dbinit` utility to overwrite the existing ObjectServer data files, if they are present.

It is recommended to run the `nco_osreport` utility on a regular basis as part of a comprehensive backup régime.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Recovering an ObjectServer

An ObjectServer can be recovered in a number of ways utilising the different tools outlined in this section. In the event of an ObjectServer outage, it is recommended to attempt the following steps (in order) to recover it.

- **Try to restart the ObjectServer**

  If the ObjectServer has stopped, it may be because the process has been inadvertently killed. First, check the ObjectServer log file for any errors that may have caused the outage. Next, after any remedial actions have been taken, restart the ObjectServer. Note that the ObjectServer may have already been restarted by the Netcool Process Agent if it is running under PA control.

- **Delete the .chk and .log files**

  If the ObjectServer will not start normally, try deleting any .chk and .log files present in the ObjectServer's db directory as one of these may be corrupted and hence may prevent the ObjectServer from starting.

- **Restore master_store.tab and table_store.tab files from backup**

  If the previous steps have failed, it may be that the ObjectServer's database files are corrupted. In this case, then next step would be to restore the `master_store.tab` and `table_store.tab` files from a backup. Simply replace the existing ones with the backups and then attempt to start the ObjectServer.

- **Rebuild the ObjectServer from nco_osreport SQL files**

  If the previous steps have failed, the nco_dbinit utility can be used to rebuild an ObjectServer's configuration using the SQL files generated via the `nco_osreport` utility. If the `nco_osreport` utility is run after each time a configuration change is made, then the configuration contained within these files should be up-to-date. See the previous subsection for information on how to do this.

- **Rebuild the ObjectServer from the original solution delivery SQL files**

  If the previous steps have failed, the ObjectServer can be rebuilt using the original solution delivery SQL files and `nco_confpack` packages, where present. The section entitled: *Solution delivery*, outlines how ObjectServer configuration for an IBM Netcool solution can be delivered via SQL files and `nco_confpack` packages, if needed. These files can be used to rebuild an ObjectServer to its initial configuration. Note that this option assumes that the solution delivery files have been kept up-to-date, per the guidelines.

Resynchronising a newly restored ObjectServer

After an ObjectServer has been recovered, the next step is to resynchronise its data contents with the rest of the system so that it may resume normal service.

Note: If all cases, if an ObjectServer is restored from backed up `master_store.tab` and `table_store.tab` files, it is possible operators may see the appearance of old events from the time that the backup was taken. Restoring the ObjectServer via the original SQL solution delivery files or the `nco_osreport` SQL files will avoid the reappearance of old events.

COLLECTION LAYER OBJECTSERVER

An ObjectServer running at the Collection layer runs in isolation in that it does not have a connected backup counterpart. In the event of a catastrophic failure that has resulted in the ObjectServer being restored from a previous backup, there is no counterpart from which it can resynchronise. If there were any Probes connected to the ObjectServer prior to its outage, those Probes would have failed over to the counterpart and replayed any event data that was not forwarded to the Aggregation layer. Hence it is not necessary to restore the data contents of a Collection ObjectServer in the case of an outage.

AGGREGATION LAYER OBJECTSERVER

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

An ObjectServer running at the Aggregation layer runs as part of a primary-backup pair. In the event of a catastrophic failure of one of the members that has resulted in the ObjectServer being restored from a previous backup, the ObjectServer will automatically be resynchronised with its counterpart via the failover bidirectional ObjectServer Gateway.

DISPLAY LAYER OBJECTSERVER

An ObjectServer running at the Display layer runs in isolation in that it does not have a connected backup counterpart. Display layer ObjectServers, however, run as a replica of the acting Aggregation primary. In the event of a catastrophic failure that has resulted in the ObjectServer being restored from a previous backup therefore, a Display layer ObjectServer will simply resynchronise itself with the Aggregation layer and continue normal operation, if restarted.

Event flood control

Event floods (also known as event storms) in the context of this document are defined as a sudden, large influx of events into the IBM Netcool/OMNIbus infrastructure. Event flood control is about engineering automated mechanisms into an IBM Netcool/OMNIbus installation that will protect against the effects of event floods.

The topic previously covered: entitled *Monitoring row numbers* on page 86, has some overlap in so much as it involves engineering automated mechanisms to purge the system of events of lesser importance when the table row numbers go beyond pre-defined thresholds. ObjectServer tables filling up are one of the side-effects of an event flood or storm.

IBM Netcool/OMNIbus systems can also potentially become overwhelmed however when event numbers are below these thresholds and a sudden rush of events is received. For example, if IBM Netcool/Impact is configured to process each new event received, the load incurred on the ObjectServer of IBM Netcool/Impact trying to process all the new events may cause a significant performance hit, if too many events are received at once.

There are many ways to cope with a large influx of events. Some example strategies are listed below and can be used individually or combined to complement each other:

- Implement Probe rule file logic to monitor event throughput load and either discard events or redirect them to an alternative ObjectServer should the throughput breach a pre-defined threshold. An example of how to do this is included in the section entitled *Using load functions to monitor nodes* in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*.
- Install a Collection layer of ObjectServers to provide a buffering layer for events to deduplicate against. This removes the load of the numerous write locks on the Aggregation ObjectServer and instead presents a summary of events to the Aggregation layer via the Collection to Aggregation ObjectServer Gateway. This approach will only make significant difference if the incoming events are particularly prone to deduplication.
- Install functionality into the Collection layer that holds events at the Collection layer until the event flood has passed, and then only forward up events to the Aggregation layer in batches. This will provide protection for the Aggregation and Display layers so that they are not overwhelmed. Synthetic events can be generated during such times to alert users that an event flood condition has been detected and event flow restrictions are in operation.

It is recommended that some sort of flood control is engineered into any IBM Netcool/OMNIbus system to ensure that an event storm does not down bring the event monitoring system. The nature of the solution will likely be constrained by various factors including business event handling policies, SLAs or other business requirements.

Note that if automated event flow restrictions are in place during an event storm, *Accelerated Event Notification* (AEN) can be used to propagate "priority" events through to the Display layer (for example, the synthetic warning events) while most events get held back. This would allow SLAs to still be met, without overwhelming the system. See the following section for more information on AEN.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: When implementing event flood control, adding such functionality to the Probe rules should always be the first point of consideration in any design. See the section entitled *Detecting event floods and anomalous event rates* in *Chapter 5* of this document for additional reading and information on event flood control from the perspective of the Probe.

Accelerated Event Notification (AEN)

You can configure IBM Netcool/OMNIbus for Accelerated Event Notification (AEN) of events that could present a risk to the system, or any other events identified by the business as being critical to their operations. The AEN system provides a means of accelerating high-priority events to help ensure that these events are displayed to operators as quickly as possible.

AEN is particularly useful in a multi-tiered IBM Netcool/OMNIbus environment to ensure high-importance, time-critical events get accelerated through the tiers faster than the other “regular” events. With AEN implemented, events can propagate almost instantaneously from the Collection layer to the Display layer the moment they are inserted into a Collection layer ObjectServer by the Probe.

An example application of where AEN would be useful is in an event flood scenario. When an event flood situation is detected, the synthetic warning event could be sent up to the Display layer immediately to notify users of the detected event flood, even though the rest of the events are temporarily held back.

A native desktop client is also provided with IBM Netcool/OMNIbus to provide near instantaneous end-user pop-ups when certain events are received.

Note: For more information regarding how to set up AEN, see *Chapter 6. Configuring accelerated event notification* in the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

Maximum number of events in the Event List

ObjectServers can store a large number of events however there are recommended limits that should not be exceeded. Ideally an IBM Netcool/OMNIbus ObjectServer will not contain more than 100,000 standing rows at any one time. Similarly, the Event List filters should also be engineered so that the number of rows displayed in an Event List does not exceed recommended limits for the respective Event List types.

The maximum recommended number of events the IBM Netcool/OMNIbus Native Event List should display is around 50,000 rows; with around 10 columns displayed.

Similarly, the maximum recommended number of events the IBM Netcool/OMNIbus WebGUI Active Event List (AEL) should display is around 50,000 rows; with around 10 columns displayed. Additionally, if more than 25,000 rows are likely to be displayed, it is recommended to increase the client JRE Xmx property to 256 MB.

The maximum recommended number of events the IBM Netcool/OMNIbus WebGUI Event Viewer (new in 8.1) should display is around 20,000. This restriction is set by the `eventviewer.pagesize.max` property in WebGUI server `server.init` file. Setting this property to a value of -1 effectively disables it.

Note: These recommended maximum limits are provided as a *guideline* of what not to exceed and do not constitute any sort of guarantee, since there are many dependencies — for example: latency of network between client and server; hardware characteristics of client machine. Exceeding these limits *may* result in a degradation of client performance, although this may also happen sooner due to these other contributory factors. Conversely, client performance *may* also run satisfactorily *above* these limits. Only thorough performance testing will yield the true limits of any given system in terms of the number of events that can be displayed for each client type.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Setting up support for multitenancy

In some cases, a single deployment of Netcool/OMNIbus may contain events that belong to different "customers" or tenants. In this case, each customer needs to be restricted to only see events that belong to them. A typical example of this is where an IBM Netcool/OMNIbus system is deployed by a Managed Service Provider (MSP) that provides event management for multiple customers. The MSP wants to provide event views to each of its customers however each customer should only be allowed to see its own events.

Traditionally, this has been done in Netcool/OMNIbus using specifically constructed restriction filters: a group is created, users are added to the group, and events are assigned to that group. Finally a restriction filter is created for the group specifically the set of events the group has access to. For example:

```
CREATE RESTRICTION FILTER wcfilter ON alerts.status WHERE OwnerGID = 50;
ALTER GROUP 'widgetcom' ASSIGN RESTRICTION FILTER wcfilter;
```

Although this works, it does incur overhead in that a separate restriction filter needs to be set up for each group and then assigned to the respective group.

New functionality introduced in Netcool/OMNIbus version 8.1 provides a function "user_in_group" that simplifies the implementation of multitenancy. Instead of creating a new restriction filter for each group, only one generic restriction filter needs to be created:

```
CREATE RESTRICTION FILTER groupfilter ON alerts.status WHERE
   user_in_group(current_userid(), OwnerGID) = TRUE;
```

This newly created restriction filter restricts the user to access only those events assigned to a group to which that user is a member. This generic restriction filter can then be applied to any group to which the target users are all members. For example:

```
ALTER GROUP 'Normal' ASSIGN RESTRICTION FILTER groupfilter;
```

With this in-place, an administrator can control the events users see simply by modifying group membership, assuming the events are being assigned to groups (ie. `OwnerGID` is set).

Note: The default group "Public" (ie. OwnerGID = 0) includes all users hence any events left uncategorised to a specific group will be seen by all users, even with this generic restriction filter in-place. Any events an administrator does not want users to see in this case should be assigned to an administrative group (eg. OwnerGID = 1).

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 5 Probes

Netcool Probes are the primary source of events for IBM Netcool/OMNIbus. This chapter contains best practice guidelines that apply to Probe configuration.

Probe configuration file locations

This section contains best practices for Probe configuration file management.

Back up Probe configuration files before modification

Ensure that a backup of any default Probe configuration file is always taken before any modifications are made. This ensures that the original default file is always available for reference, should it be needed.

A suggested naming convention for the original file is to append the copied filename with a `.orig` file extension. For example:

`cp tivoli_eif.rules tivoli_eif.rules.orig` (UNIX)

`copy tivoli_eif.rules tivoli_eif.rules.orig` (Windows)

Thereafter, if any updates are made to the *"live"* file, a further backup copy should be taken before making any changes. This will allow a rapid roll-back, if required.

A suggested naming convention for the backup copy filenames is to append them with a `.<date>` file extension. For example:

`cp tivoli_eif.rules tivoli_eif.rules.20110921` (UNIX)

`copy tivoli_eif.rules tivoli_eif.rules.20110921` (Windows)

Create custom Probe configuration file subdirectories

The default location for Probe configuration files is:

`$OMNIHOME/probes/<arch>/` (UNIX)

`%OMNIHOME%\probes\win32\` (Windows)

It is recommended that copies of the default files are copied to a separate directory for modification and use, just as is done for Netcool Gateway configuration files. For example:

`mkdir $OMNIHOME/etc/probes/` (UNIX)

`mkdir %OMNIHOME%\etc\probes\` (Windows)

Probe configuration files can be further grouped and organised by creating a further subdirectory for each Probe. For example:

`mkdir $OMNIHOME/etc/probes/tivoli_eif\` (UNIX)

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

`mkdir %OMNIHOME%\etc\probes\tivoli_eif\` (Windows)

The default configuration files can then be copied from the default location to this live, *"working"* location ready for customisation and use. This technique will make it easier to package up customisations as well as keep customisations separate from original, default configuration files.

Note: If you are storing the *"live"* configuration files in alternative locations to the default locations, you will need to add those alternative locations to the Probe configuration. For example, the Probe's working copy of the properties file would need to be referenced in the IBM Netcool Process Agent configuration file process command, and the Probe's working copy of the rules file would need to be referenced in the Probe's properties file.

Generic properties

This section contains key best practice points in relation to generic properties that apply to all Probes.

Probe peering recommended where appropriate

For resiliency, it is generally recommended that Probes be deployed in master-slave pairs. If a master Probe fails for whatever reason, the slave Probe takes up the role of master and continues until the master is restored.

Note that whether to use Probe peering depends on the type of Probe being deployed. A Probe that monitors a single event source that is collocated on the same system as the event source would likely not benefit from being peered with a slave Probe. In such a case, if the target system is down, the collocated Probe will be down too, including any slave Probe. Even if the slave Probe was remotely located, it would not be able to connect to the event source anyway, and so would be redundant. For example, a SCOM Probe deployed on a standalone SCOM system would not benefit from having a slave present.

The SNMP Probe and the Tivoli EIF Probe however (ie. "listening" Probes) are good examples of where Probe peering is appropriate and provides resiliency. Note that the managed entities must be configured to send events to both the master and slave Probes however to benefit.

Note: When Probe peering is configured, the slave Probe will discard the contents of its buffer each time it receives a heartbeat from the master. If the master is not receiving any events for some reason (for example, due to a network fault) but the slave is, the slave will still discard its events so long as it receives the heartbeat from the master. Consideration must be given to this sort of scenario therefore when designing an IBM Netcool architecture. In some cases, it may be necessary to install two standalone Probes, and potentially receive twice the number of events, to avoid missing events altogether.

See the table in the section entitled: *Generic properties* on page 101 for details on the specific properties that configure Probe peering and the recommended values.

Circular store and forward mode for Probe failover/failback

Store and forward is a feature of Probes that allow them to buffer events if they become disconnected from the ObjectServer. When a Probe can reconnect to an ObjectServer, it will send the buffered events through to the ObjectServer before resuming normal operation.

There are scenarios however where, although a Probe has successfully transferred a batch of events to an ObjectServer, the events may not be seen by an operator in a timely fashion. For example, a Probe may have sent a batch of events to a primary Collection ObjectServer but that ObjectServer goes down before it forwards these events up to the Aggregation layer. Until the primary Collection ObjectServer is restored, these events will remain unseen by operators.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

When a Probe runs in circular store-and-forward mode, it stores a rolling history of the last *N* seconds' worth of events in local files. If a Probe is disconnected from the ObjectServer and subsequently reconnects, it will replay the historical events (ie. the contents of the files) before resuming normal operation.

If circular store and forward is applied to the scenario above, although the events never made it from the primary Collection ObjectServer to the Aggregation layer, at least the backup Collection ObjectServer will get a fresh copy of the missed events, so that they can be relayed up to the Aggregation layer straight away.

When the primary Collection ObjectServer is eventually restored, it will attempt to send the old events up to the Aggregation layer. The default ObjectServer behaviour on both the Collection and Aggregation deduplication triggers is to discard the attempted inserts by Probes of old, duplicate events.

Note: See the deduplication triggers `col_deduplication` and `agg_deduplication` in the multitier architecture configuration files `collection.sql` and `aggregation.sql` respectively, located in: `$OMNIHOME/extensions/multitier/objectserver/` — to see how the ObjectServer deduces that an incoming Probe event is a duplicate.

Hence the circular store and forward functionality forms a best practice for the configuration of Probes and is therefore included in the section entitled *Generic properties* on page 101. Note however that other store and forward options are available and may be more suitable for use depending on individual business event management policies. This strategy and configuration merely represents a baseline configuration and is suggested as a default if no such policies are defined.

Note: For more information about circular store and forward, see the section entitled *Circular store and forward* in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*.

Enabling self-monitoring of Probes

You can configure Probes to generate *ProbeWatch* heartbeat events as a self-monitoring mechanism. By enabling Probe heart-beating, operators and Netcool Administrators alike can monitor the health and state of Probes. Automated processes can be engineered to monitor the heartbeat events and act when a heartbeat is not received by one of the Probes.

After enabling self-monitoring, the Probe can also be configured to forward statistical data to the `master.probestats` table in the ObjectServer, to be subsequently processed by a set of pre-canned triggers provided in the `$OMNIHOME/extensions/roi/` directory. Note that this extension is optional and only needs to be configured if it is required.

The property that enables Probe self-monitoring (`ProbeWatchHeartbeatInterval`) is set to 0 (ie. zero) by default — which is its disabled state. It is recommended to use a non-zero value for this property for all Probes. See the next section for recommended property values.

For more information about enabling self-monitoring of Probes and statistics gathering and analysis, see the following sections in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*:

- *Enabling self monitoring of Probes*
- *Configuration setup for self monitoring of probes*
- *IBM Netcool/OMNIbus configuration files for the self monitoring of probes*
- *Configuring probes for self monitoring*

Generic properties

The following table outlines some key generic properties that should be considered whenever deploying an IBM Netcool Probe.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Probe property	Comment	Recommended value
Server ServerBackup	There are two alternative ways to specify how a Probe will connect to the ObjectServers: • Specify only the *virtual* name for an ObjectServer pair in the Server property — for example: COL_V_1 (made up of COL_P_1 and COL_B_1) or AGG_V (made up of AGG_P and AGG_B); • Specify the *actual* name for the primary ObjectServer in the Server property — for example: COL_P_1 or AGG_P — and the *actual* name for the backup ObjectServer in the ServerBackup property — for example: COL_B_1 or AGG_B. Note that if using the first option that uses the virtual name in the Server property, the ServerBackup property should not be used.	See comment section
NetworkTimeout	If a host running a primary ObjectServer gets its network cable pulled out, any connected processes may have to wait 10 minutes or more for the TCP/IP layer to time out, depending on the operating system setting. In the case of a Probe, this could mean a lengthy delay before failing over to the backup ObjectServer. To ensure the Probe fails over in a more timely manner in the event of an ObjectServer outage, this property can be used to specify the time (in seconds) to wait before timing out a TCP/IP connection and failing over or failing back. The default value for this property is 0, meaning wait for the operating system to time the connection out. A suggested value for this property is 120 (seconds).	120

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

PollServer	This property defines how many seconds to remain connected to a backup ObjectServer between attempts to fail back. A Probe knows it is connected to a backup ObjectServer because it queries the ObjectServer's `BackupObjectServer` property on initial connection. If the ObjectServer's `BackupObjectServer` property is set to TRUE, then the Probe will periodically attempt to fail back every number of seconds corresponding to the value stored in the `PollServer` property. If the ObjectServer's `BackupObjectServer` property is set to FALSE however, then the Probe remains connected to the current ObjectServer indefinitely. The default value of this property is 0, meaning it will never try to fail back to the primary ObjectServer. A value of 0 is suitable when connecting Probes to the Aggregation layer, since failback is controlled by the backup Aggregation ObjectServer. When the primary Aggregation ObjectServer is restored and the bidirectional Aggregation ObjectServer Gateway has finished its resynchronisation, the backup Aggregation ObjectServer will automatically disconnect all connected clients, including Probes, which prompts their failback. When connecting Probes to the Collection layer however, then automated failback is preferable since otherwise Probes would remain indefinitely connected to the backup Collection ObjectServer. A suggested alternative value for this property is 120 (seconds) when connecting Probes to the Collection layer.	Aggregation layer: 0 Collection layer: 120
Mode	This property along with `PeerHost` and `Peerport` is used for Probe peering. This property should be set to "master" for the master Probe and "slave" for the slave Probe.	See comment section
PeerHost	This property along with `Mode` and `Peerport` is used for Probe peering. This property should be set to the hostname of the machine where the counterpart is running.	See comment section
Peerport	This property along with `Mode` and `PeerHost` is used for Probe peering. This property should be set to the port of the machine where the counterpart is running. A suggested setting for this property is to simply use the default of 9999. Note that if the default value is used, the property does not need to be explicitly defined.	Use default: 9999
BeatInterval	When Probe peering is set up, the slave Probe will discard any events it receives while the master Probe is active. The slave Probe knows the master is active because a properly configured master Probe will send heartbeats to the slave Probe. The interval between heartbeats is defined by the `BeatInterval` property. It is recommended this property use the default value of 2 seconds.	2

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

BeatThreshold	The slave Probe caches all the alert data it receives and discards all alert data in its cache each time it receives a heartbeat from the master instance. If the slave instance receives no heartbeat in the time period defined by the sum of the values of the `BeatInterval` and `BeatThreshold` properties (`BeatInterval + BeatThreshold`), the slave instance assumes that the master is no longer active, and forwards all alerts in its cache to the ObjectServer. The timeout period while waiting for heartbeats is 1 second. Therefore, there can be a maximum delay of (`BeatInterval + BeatThreshold + 1`) seconds before the slave Probe forwards its cached alerts. All alerts in the cache are sent. The slave Probe continues to forward all alerts until it receives another heartbeat from the original master Probe at which time the slave Probe will stop forwarding its events. Note that the `BeatInterval` setting that is defined for the master instance takes precedence. The slave instance ignores its local `BeatInterval` setting. See the section entitled *Peer-to-peer failover mode for probes* on page 14 of the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide* for more information on `BeatInterval` and `BeatThreshold`.	1
StoreAndForward	To invoke the use of circular store and forward, this property should be set with a value of 2.	2
SAFPoolSize	It is important to ensure the Probe has a large amount of buffer space available to store historical events, so that no events are lost. To ensure the Probe has a large pool of files to store events, it is recommended this property be set initially to a value of 10.	10
MaxSAFFileSize	It is recommended this property be set initially to a value of 10,000,000 bytes (ie. 10 MB).	10000000
RollSAFInterval	This property defines the number of seconds' worth of historical event data the Probe retains on a rolling basis. It is recommended to use the default setting of 90 (seconds) as per the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*.	90
AutoSAF	When this property is enabled, the Probe will automatically store events if its connection to the ObjectServer is lost. This property should be enabled (ie. set to 1) for all types of Probes.	1
ProbeWatchHeartbeatInterval	By setting this property to a non-zero value, it both enables Probe heart-beating and specifies the regularity of the heartbeat events. It is recommended this property be initially set to a value of 60 (seconds).	60
Buffering	It is recommended to enable buffering to increase event throughput through Probes hence it is recommended to set this property to 1 (ie. enabled). The default setting is 0.	1

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

BufferSize	The BufferSize specifies the number of events the buffer should hold before the Probe sends the events to the ObjectServer. It is recommended to set the BufferSize to 100 (events). This property setting in conjunction with the BufferFlushInterval (below) has been found to yield excellent overall performance. The default setting is 10.	100
BufferFlushInterval	The BufferFlushInterval is the maximum time (in seconds) to wait before the Probe sends the buffer contents off to the ObjectServer if the BufferSize number of buffered events has not been reached. It is recommended to set the BufferFlushInterval to 1 (second). The default setting is 0 which disables this function.	1

Note: See the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide* for more information on these generic properties plus the other generic Probe properties. Information relating to specific Probe properties can be found in each Probe's respective documentation.

dumpprops property

To inspect a Probe's current properties settings and to see all the available runtime options, simply run the Probe with the `dumpprops` switch as in the following example:

```
$OMNIHOME/probes/nco_p_syslog -dumpprops
```

Note: The output will consider any properties set in a Probe's properties file.

Running more than one instance of a Probe on the same host

It is possible to run more than one instance of a Probe on the same box — for example: the Generic Log File (GLF) Probe may have multiple instances running on the same box — each one monitoring a different log file.

When running multiple instances of the same Probe on the same host machine, care should be taken to ensure the multiple instances of the Probe do not interfere with the operation of each other, and that different properties are specified for each instance of the Probe, where appropriate.

Every instance of each Probe should have a unique name and have its own properties file. It is also common that each Probe instance will have a different rules file, depending on the Probe type. The properties file each Probe instance should be specified on the corresponding Process Agent configuration file entry using the *propsfile* property — for example:

```
$OMNIHOME/probes/nco_p_glf -propsfile $OMNIHOME/etc/probes/glf/glf1.props
```

The rest of the Probe properties can be included in the properties file but should start with the following:

- `Name` — unique name for the Probe;
- `RulesFile` — unique rules file for the Probe.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: In addition to the properties above, it is likely that there will also be Probe-specific properties that need to be specified differently for each Probe instance. The relevant Probe documentation should be consulted in each case as to what the specific Probe properties are and how to set them.

EXAMPLE:

The Netcool Administrator at Widgetcom needs to have two instances of the GLF Probe running on one of the UNIX servers to monitor two different log files. She starts by creating two copies of the default properties files and names them as follows:

- `$OMNIHOME/etc/probes/glf/glf1.props`
- `$OMNIHOME/etc/probes/glf/glf2.props`

In the first properties file: `glf1.props`, the following properties are specified:

```
Name                    : "GLF1"
RulesFile               : "$OMNIHOME/etc/probes/glf/glf1.rules"
```

In the second properties file: `glf2.props`, the following properties are specified:

```
Name                    : "GLF2"
RulesFile               : "$OMNIHOME/etc/probes/glf/glf2.rules"
```

Two copies of the default GLF Probe rules files are copied to the specified locations and then updates made to the files, as required. Finally, two instances of the GLF Probe are configured to run in the Netcool Process Agent of the host machine: one set to load *glf1.props* and the other to load *glf2.props*.

Probe rules files

An IBM Netcool Probe is usually the first point of entry for an event into an IBM Netcool system and a Probe's rules file is the first point at which incoming events are analysed, enriched via lookup tables, discarded, and otherwise processed and prepared, ready for insertion into an ObjectServer. A Probe rules file therefore has a significant role to play in the correct and efficient handling of incoming event data.

In addition to accuracy and correct handling, performance is of particular importance. Some Probes handle up to 1,000 or more events per second, especially during peaks, and so need to be written to be as efficient as possible. A microsecond saved here and there for each processed event can make a big difference to the overall performance of a Probe when it is processing millions of events per day.

When running in DEBUG mode, Probes will output the time taken to process each event (in microseconds and millionths of a second on UNIX, and thousandths of a second on Windows). This is a useful way to test out the *relative* efficiency of a Probe rules file in a development or test environment, to ensure that the rules file engine isn't becoming the bottleneck of a Probe. For example, in order to process 1,000 events per second, the rules file needs to be processed in under 1 millisecond (ie. in 1/1000th of a second).

Finally, common coding concepts such as appropriate use of whitespace, comments, and indentation are all essential to ensure long-term maintainability of Probe rules files.

This section covers all these principles and they form the basis for Probe rules file best practice.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: For further reading on Probe rules file development guidelines, see the section entitled *Rules file development* for guidelines on Probe rules file development in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*.

Use of comments

Probe rules files should contain extensive commenting, both in a header section as well as in-line in the code. Having extensive commenting in-line in the code is essential. As rules files get updated over time, in-line comments become increasingly important so that an engineer carrying out modifications is in no doubt about what existing code does.

Additionally, custom code should be demarcated from default code to make it easier to read. Rules file comments and other dividers can be used to achieve this.

EXAMPLE:

Widgetcom have added a new field to their ITM installation and want it passed through to the IBM Netcool/OMNIbus infrastructure via the EIF Probe. The additional field contains key information needed by IBM Netcool/Impact for event enrichment. Not every event coming from ITM will contain data in this field, but where it does, leading asterisk characters should be trimmed off the contents before being passed to the `DeviceCode` field in the ObjectServer.

The Netcool Administrator adds the following Probe rules file code to the end of the Tivoli EIF Probe rules file, and includes the Probe's rules file in the solution delivery package:

```
####################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 19-Sep-2011
# This section of Probe rules file contains Widgetcom customisations
# to handle the incoming $device_code field token.
# If the token exists, it should have any leading asterisk
# characters trimmed off and then be assigned to the ObjectServer
# field DeviceCode.
####################################################################

# CHECK IF INCOMING TOKEN $device_code IS PRESENT IN INCOMING EVENT
if (exists($device_code)) {

   # CHECK IF $device_code HAS LEADING ASTERISK CHARACTERS
   if (regmatch($device_code, "^[\*]+.*")) {

         # TRIM OFF ANY LEADING ASTERISK CHARACTERS
         $device_code = extract($device_code, "^[\*]+(.*)")
   }

   # ASSIGN $device_code TO OBJECTSERVER FIELD DeviceCode
   @DeviceCode = $device_code
}
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

NOTES:

- Just as for ObjectServer triggers, custom Probe rules file code additions should include who modified the file and when they did it, plus a detailed description of what the code addition does and how.
- In-line code comments in this example are shown in CAPITAL LETTERS (except for field names), to help them stand out. Conventions such as these are optional, however the use of any conventions should be *consistent* so that code is easier to read.
- Before the regmatch is done on the token `$device_code`, the `exists` function is called first to ensure the token exists before it is accessed. Attempting to access non-existent tokens will generate errors in the Probe log file.
- The code uses consistent indentation conventions. Tab characters are used where appropriate.

Note: See the section entitled *Standardisation of layout and formatting* in *Chapter 4 ObjectServers* for a commentary on the standardisation of trigger code layout and formatting. The same principles of defining and consistently applying conventions also apply to Probe rules file code, as it does with any code that is produced as part of the solution delivery.

Location of customisations within the rules file

To help demarcate custom rules file code from default code, it is recommended to place custom code at the bottom of the file where possible. This is not always possible however — for example: array and lookup file declaration statements must go at the top; custom *includes* in the SNMP Probe rules file must go in-line in the place where the enterprise ID is evaluated.

In all cases, custom rules file code should be clearly demarcated using dividers — for example: a row of hash symbols — as in the example given in the previous section.

Use of indentation and white space

White space and indentation should be consistently used in Probe rules file customisations to make code customisations clearer and easier to read. It matters less about *what* conventions are used, if they are sensible and applied *consistently*. Applying this best practice results in a more professional piece of rules file code and aids readability.

Sample conventions used in the Probe rules file example above include:

- A blank line is left between the main description and the first line of code to aid readability.
- Nested code fragments (for example, statements that are nested within an IF statement construct) are indented by one tab character more than the IF statement itself. Further nesting is indented by adding more tab characters.
- The opening parenthesis for each IF statement appears on the same line as the condition(s) and is separated by one space character.
- The first line after each IF statement (or any other nested construct) has a blank line to aid readability.
- Arguments passed in function calls (eg. `regmatch`) are each separated by a comma and a space character.
- A blank line is left after each closing parenthesis to aid readability.

Code efficiency

As mentioned in a previous section, an optimisation in a Probe rules file that saves the smallest fraction of a second can have a significant positive impact on a Probe's overall performance, especially where a

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Probe is processing large numbers of events. Every effort should be made to make Probe rules as efficient as possible.

The following list provides some tips on Probe rules file efficiency:

- Never use a less efficient statement (eg: a call to the `regmatch` function, a regular expression match) when a more efficient alternative exists (eg: a call to the `match` function, a direct match).
- Do not use an IF-ELSEIF construct where a SWITCH or CASE statement could be used instead. SWITCH and CASE statements are more efficient.
- Engineer code so that the fewest number of lines of code will be executed for each event. Conditional constructs can be used to filter events through different sections of the code.
- Limit the use of the FOREACH statement on large arrays of data as this can cause performance bottlenecks.

Use of include files

The INCLUDE statement is a convenient way to include another rules file into the current rules file at the point that the INCLUDE statement appears.

INCLUDE files are useful when:

- A section of rules file is used in more than one place in a rules file. Rather than duplicate the code block multiple times in the rules file (thereby increasing maintenance overhead), put the code into a separate file and simply "include" it from multiple places. This use is analogous to the use of procedures in the ObjectServer — in that commonly used code blocks are partitioned off into modules that can be referenced from multiple places.
- More than one Probe uses a section of the rules file. The rules file code block can be put into a separate file and referred to by many Probes' rules files, either on the same file system or remotely obtained via an HTTP server.

Note: For more information about the INCLUDE statement, see the section entitled *Embedding multiple rules files in a rules file* in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide*.

Use of details

The DETAILS statement within a Probe rules file causes the tokens specified to be passed along with the event to the ObjectServer. This is a normal practice with the SNMP Probe where specific trap *varbinds* are sent to the ObjectServer with the event, depending on the event type, to provide additional key information to the operator. These event details are then visible to the Event List user on the *Details* tab of the event information window.

Note: New functionality in Probes and the ObjectServer make provision for an alternative, more efficient mechanism to make Probe data tokens available within the ObjectServer. This involves the use of the `nvp_*` functions in both Probe rules and ObjectServer SQL used in conjunction with the `ExtendedAttr` field. Both *details* and *journals* have a performance hit on an IBM Netcool/OMNIbus system and so using the new mechanism over details is the preferred method to use, whenever possible. See the section entitled *Name-value pair (NVP) functions and ExtendedAttr* later in this document for advice on the use of this new functionality.

EXAMPLE:

Widgetcom is using the Generic Log File Probe to monitor the log file of a proprietary application. The log file being monitored contains one event per line and contains a variety of different types of event. The type of event is stored within token `$5`. For all events, the contents of tokens: `$6`, `$7`, and `$8` — should be combined and stored in the `@Summary` field. For events that are of type "ALERT", a number of additional tokens: `$9` thru `$18` — need to be made available to the Event List operator, but are too large to fit in a single field.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Netcool Administrator carries out some analysis of the ObjectServer event data and finds that the events of type ALERT make up only a small proportion of the total number. The Netcool Administrator therefore elects to provide the additional information for ALERT events to users via the details functionality. The operators can inspect this additional information simply by double-clicking the event and selecting the *Details* tab. The following Probe rules code is added to the end of the Generic Log File Probe rules file, which is then included in the solution delivery package:

```
#####################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 19-Sep-2011
# This section of Probe rules file has been added to provide
# additional information to operators via the tokens $9 thru $18.
# The additional tokens will be included as details if the current
# event is of type ALERT.
#####################################################################

# CHECK IF TOKEN $5 EXISTS
if (exists($5)) {

   # CHECK IF CURRENT EVENT IS OF TYPE ALERT
   if (match($5, "ALERT")) {

         # INCLUDE TOKENS $9 THRU $18 AS DETAILS
         details($9, $10, $11, $12, $13, $14, $15, $16, $17, $18)
   }
}
```

NOTES:

- Before checking the value of the token `$5`, a check is first made to ensure the token exists within the current event incoming data stream.
- A direct match is made to check the contents of the token `$5`. This is more efficient than a regular expression match.

The DETAILS statement is also commonly used in Probe rules file development, typically by including the following statement on the last line of a Probe rules file:

```
details($*)
```

This causes the Probe to send *all* event tokens along with the event to the ObjectServer, which can be useful for debugging purposes in a development environment.

Warning: The use of `details($*)` should *not* be used by any Probe in a production environment due to the load it will invariably place on the ObjectServer and Gateway infrastructure. This is a strict best practice principle. It is permitted however to include specific details – for example: `details($token1, $token2)`

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The practice of including *specific* details with an event (ie. as the SNMP Probe does) is not contrary to best practice, however its use should be limited to only when necessary. The use of details incurs similar degree of extra loading on the ObjectServer as journals do.

Note: See the section entitled *Monitoring row numbers* on page 86 in this document for information on table size monitoring strategies. Also see the section entitled *Manage the volume of information in the alerts.details table* in the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

Name-value pair (NVP) functions and ExtendedAttr

The NVP functions in both the Probe rules file language and the ObjectServer SQL language used in conjunction with the `ExtendedAttr` field in the ObjectServer provide an alternative method of attaching and retrieving bulk name/value pairs to events to the traditional method of using *details*.

As noted in the previous section, the use of details incurs a performance hit on the IBM Netcool/OMNIbus infrastructure. The reason the use of details incurs a performance hit is due to way that they are stored and subsequently retrieved whenever they are used.

Details are stored in the `alerts.details` table and row items in this table are linked to events in the `alerts.status` table via the `Identifier` field. When a component such as an AEL, Native Event List or Gateway selects a set of events from the `alerts.status` table, the `alerts.details` table is also scanned to locate any linked items. These table scans can be costly, particularly as the `alerts.details` table increases in size.

The NVP functionality, on the other hand, stores this additional data in a single field (ie. `ExtendedAttr`) within each event. This means that this additional data does not have to be fetched separately (incurring additional processing cost) each time an event is selected.

The cost incurred by storing additional data items within the event come in the form of an increased ObjectServer memory footprint, which would happen anyway if this additional data was being stored in the `alerts.details` table.

EXAMPLE:

Widgetcom receive events via the EIF Probe. One of the tokens received by the Probe is called `$esc_code` and contains a code which is only meaningful in the context of an event if the event is escalated. The escalation code needs to be included with the event however it only needs to be made visible to the user (by including it in the Summary field) if the event is escalated via *Widgetcom's* event escalation process.

The Netcool Administrator elects to store the escalation code for events in a NVP for use in case it is needed. The following Probe rules code is added to the end of the EIF Probe rules file, which is then included in the solution delivery package:

```
######################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 19-Sep-2011
# This section of Probe rules file has been added to store the
# escalation code in a NVP where it exists.
######################################################################

# CHECK IF ESCALATION CODE TOKEN EXISTS
if (exists($esc_code)) {

   # STORE ESCALATION CODE IN NVP FOR LATER USE
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
  @ExtendedAttr = nvp_add(@ExtendedAttr,"esc_code", $esc_code)
}
```

Next, the Netcool Administrator creates a database trigger that retrieves the escalation code and prefixes it to the `Summary` field when an event is escalated.

The following SQL is added to the *Widgetcom* solution delivery SQL file:

```
-------------------------------------------------------------------
-- CREATE THE ESCALATION CODE PREFIX TRIGGER
CREATE OR REPLACE TRIGGER prefix_esc_code
GROUP widgetcom_triggers
PRIORITY 1
COMMENT 'TRIGGER prefix_esc_code
Last modified by: J Smith (jsmith) - Netcool Admin - 21-Sep-2011
This trigger detects the moment an event is escalated and
subsequently prefixes the event escalation code to the Summary
field as a result - if available.'
BEFORE UPDATE ON alerts.status
FOR EACH ROW
WHEN get_prop_value('ActingPrimary') %= 'TRUE'
declare
  evt_esc_code char(12);

begin

  -- DETECT EVENT ESCALATION
  if (old.EscalationState = 0 and new.EscalationState = 1) then

        -- CHECK THAT ESCALATION CODE EXISTS FOR THIS EVENT
        if (nvp_exists(ExtendedAttr, 'esc_code')) then

              -- RETRIEVE ESCALATION CODE FROM NVP
              set evt_esc_code = nvp_get(ExtendedAttr, 'esc_code')

              -- PREFIX Summary FIELD WITH ESCALATION CODE
              set old.Summary = evt_esc_code + ' ' + old.Summary;
        end if;
  end if;
end;
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
go
```

NOTES:

- The trigger is written in a generic way in that it will potentially perform the prefix action for any type of event where the escalation code exists in the NVP set. This means this functionality could be extended to any of the other Probes simply by creating this NVP in their respective rules files.

Note: For further reading on the use of the NVP functions, see the section entitled *String functions* in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide* for use within Probe rules and the section entitled *Functions* in the *IBM Netcool/OMNIbus 8.1 Administration Guide* for use within ObjectServer SQL.

Netcool Knowledge Library (NcKL)

The SNMP Probe (formerly called the *"Mttrapd"* Probe) is used to receive events from any type of device that can send a trap. The task of decoding the enterprise ID, specific trap ID and unpacking the "varbinds" stored in the trap into a human readable format is done in Probe rules files.

The Netcool Knowledge Library (NcKL) is a pre-canned set of rules file written for the SNMP Probe that includes rules for handling traps from a wide variety of vendors. This means that engineers installing and configuring the SNMP Probe potentially don't have to write rules files to handle traps from included devices, as they can simply use the ones provided in the NcKL, saving time and effort.

The use of the Netcool Knowledge Library with the SNMP Probe is considered a best practice when managing devices included in the library. The Netcool Knowledge Library can be downloaded from the IBM Passport Advantage web site:

https://www.ibm.com/software/passportadvantage/pao_customer.html

The latest version of the Netcool Knowledge Library at the time of publication is version 4.7. The package to download is as follows:

```
Netcool/OMNIbus 8 Plus Netcool Knowledge Library (NcKL) Enhanced
Probe Rules V4.7 Multiplatform English (CNIA8EN )
14 Mar 2017

Netcool Knowledge Library (NcKL) Enhanced Probe Rules V4.7
14 Mar 2017
```

For full downloading and installation instructions, refer to the following link:

http://www.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/probes/nckl/wip/concept/nckl_intro.html

Note: IBM personnel may download the NcKL from the *Xtreme Leverage* intranet site.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

MIB to rules file conversion

Although the Netcool Knowledge Library (NcKL) contains a vast selection of pre-written rules files to handle traps from a wide variety of vendors, there will often be trap devices in an IBM Netcool deployment that are not covered by NcKL.

IBM provides a utility called the *IBM Netcool MIB Manager* which is an IBM Eclipse based utility that can be used to convert SNMP MIB files into Netcool Probe rules files. IBM Netcool MIB Manager is the successor to the utility formerly known as *mib2rules* and supports the latest NcKL export formats.

The IBM MIB Manager ships with IBM Netcool/OMNIbus 7.4 onwards and is launched by running the following command:

```
$OMNIHOME/bin/nco_mibmanager
```

Detecting event floods and anomalous event rates

Event flood control was introduced in *Chapter 4* in the section entitled *Event flood control* and touched on the concept of implementing Probe rule file logic to monitor event throughput load.

Note: The terms *"event flood"* and *"event storm"* can be used interchangeably.

Having visibility of a Probe's loading and state allows an IBM Netcool Administrator to respond more quickly to outages and also provides the potential for Administrators to avert outages by taking preventative action.

Note: See the section entitled *Detecting event floods and anomalous event rates* in the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide* for more information on various strategies for detecting event floods and anomalous event rates.

It is recommended that some sort of flood control is engineered into any IBM Netcool/OMNIbus system to ensure that an event flood does not down bring the event monitoring system. The nature of the solution will likely be constrained by various factors including business event handling policies, SLAs, or other business requirements.

Note: Also see the section entitled *Event flood control* in *Chapter 4 ObjectServers* of this document for additional reading and information on event flood control from an overall perspective.

Probe rules file development tips and tricks

This section contains useful techniques that can be used when developing Probe rules files. They are not in themselves "best practices" — rather they are recommended methods for achieving the respective tasks outlined by each one and may be used if required.

Debugging Probe rules files

The Netcool/IDE is a useful GUI rules file editor and debugger and is available from:

https://developer.ibm.com/itoa/resources/netcool-ide/

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Another way to help with debugging rules files is to add LOG statements to output the values of variables at certain points within your Probe rules file. This helps confirm any assumptions you are making about what values they hold at that moment — for example:

```
log(WARNING, "Hit point A in rules, $wibble = [" + $wibble + "]")
```

RawCapture output for unparsed events

The `RawCapture` property may be set dynamically within the rules file to write out the current event into the raw capture file. This is appropriate whenever an event is encountered by the Probe rules file that does not follow the expected format. The following is an example of how this technique can be used.

Near the top of the rules file (after the targets, tables, and arrays) set the following:

```
# Turn %RawCapture property off
# Note that this will override setting in the properties file
# or command line
%RawCapture = 0
```

Then, where you get to a point in the rules file that you don't have rules to deal with an event — for example: in the default case of the `switch($enterprise)` in the NcKL rules file:

```
switch($enterprise) {
   ...
   default:
         log(DEBUG, "<<<<< (snmptrap.rules) Enterprise ID not found
in any include file. >>>>>")
         @Summary = "No Rules Found for Enterprise ID: " +
               $enterprise + " (see details)"
         @Severity = 2
         @Type = 0
         %RawCapture=1
}
```

This will provide raw capture output only for traps that do not have specific rules written to handle them. This is suitable for use in a production environment and it will help capture information about traps for which rules have not yet been defined.

regmatch() before extract()

Before using the `extract` statement in a Probe rules file, always perform a `regmatch` first to make sure that the `extract` function will succeed. If it does not match, then some contingency code can be executed instead — for example: logging an error message or switching on raw capture mode for the current event:

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
# CHECK IF $foo MATCHES THE EXPECTED FORMAT:
# REF: BLAH-123
if (regmatch($foo, "^[A-Z]*: BLAH-(.*)$")) {

   # EXTRACT NUMERICAL COMPONENT IF MATCHES EXPECTED FORMAT
   @AlertKey = extract($foo, "^[A-Z]*: BLAH-(.*)$")

} else {

   # ELSE LOG ERROR MESSAGE THAT IT IS NOT IN THE CORRECT FORMAT
   log(ERROR, "Data not in expected format: [" + $foo + "]")

   # ENABLE RawCapture SO TOKENS CAN BE EXAMINED LATER
   %RawCapture = 1
}
```

NOTES:

- The same regular expression string can be used for both the call to `regmatch` and `extract` (including the extraction parentheses).

Obtaining sub-second timing information in the rules file

The rules file language provides no direct call to get sub-second timing information, but you can obtain this information via the `updateload()` function call and extract it manually. Consider the following Probe rules file code fragment:

```
$timer = "2.2"
$timer = updateload($timer)
log(DEBUG, "Value: " + $timer)
```

This will produce something like the following in the Probe log file:

```
2011-09-01T12:55:06: Debug: D-UNK-000-000: Value: 2.2
1314878106.64880
```

Calling the `updateload()` function on an empty input string (ie. one with just the window size) gives you the result of the `gettimeofday()` function, which includes the current time second and microsecond values. The above example gives the following result:

```
Value: 2.2 1314878106.64880
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

From the above result, the current UTC time is 1314878106 seconds and 64880 microseconds.

Note: Remember to reset the variable (ie. `$timer = "2.2"`) before calling the `updateload()` function again, otherwise values are not guaranteed to be updated.

You can also use the `updateload()` function to measure time-specific operations within the rules file language. The following example calls the the `updateload()` function both before and after a call to the `gethostname()` function, which causes the elapsed time between the two `updateload()` function calls to be appended to the variable named `$timer`:

```
$timer = "2.2"
$timer = updateload($timer)
$hostname = gethostname("192.168.1.123")
$timer = updateload($timer)
log(ERROR, "Timing=" + $timer)
log(ERROR, "Hostname=" + $hostname)
```

The above example produces something like the following in the Probe log file:

```
2011-10-05T12:51:48: Error: E-UNK-000-000: Timing=2.2
1317815508.849179 127603
2011-10-05T12:51:48: Error: E-UNK-000-000:
Hostname=alexsmachine.example.com
```

The third value in the `$timer` string (ie. 127603) is the time in microseconds (ie. in millionths of a second) between the two `updateload()` function calls, and represents the time taken to perform the DNS lookup (ie. the call to the `gethostname()` function). 127603 microseconds is roughly 1/8th of a second.

Use regreplace() to clean up tokens/strings

The `regreplace()` function provides regular expression matching and substitution for tokens and strings and can be used in a variety of ways. Below are some examples:

- Remove all occurrences of $, ! or ' in any of the tokens:

```
foreach(x in $*)
{
   $x = regreplace($x, "[$!']", "")
}
```

- Replace multiple spaces with a single space:

```
foreach(x in $*)
{
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
   $x = regreplace($x, " +", " ")
}
```

Using the update() function

The function call:

```
update(@SomeField, FALSE)
```

...does not prevent a field from being updated by an ObjectServer reinsert (ie. deduplication) trigger, it merely cancels a previous occurrence of the function call:

```
update(@SomeField)
```

... or:

```
update(@SomeField, TRUE)
```

If the deduplication trigger contains:

```
set old.SomeField = new.SomeField;
```

...then that field will be updated regardless of what is in the rules file.

If you want to *conditionally* update a field, do not include it in an ObjectServer trigger, and instead control the update logic completely in the rules file.

Alternatively, add a flag that can be tested in the trigger to determine whether it should be updated. For example, the Probe rules file contains the following:

```
if ( <some-condition> )
{
   @UpdateSomeField = 1
}
```

...and the reinsert trigger contains the following:

```
   if (new.UpdateSomeField = 1) then

         set old.SomeField = new.SomeField;
   end if;
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 6 Gateways

An IBM Netcool Gateway provides a conduit between two ObjectServers, or between an ObjectServer and a third-party target: a database or a ticketing system, for example. This chapter contains best practice guidelines that apply to Gateway configuration.

The primary use case of a unidirectional ObjectServer Gateway is to connect two tiers in a multitiered IBM Netcool/OMNIbus environment, or for providing a unidirectional flow of events to an external IBM Netcool/OMNIbus system. In both cases, race conditions are typically not an issue since the flow of data is restricted to one direction only. The primary use case of a bidirectional ObjectServer Gateway is to connect a failover pair of ObjectServers.

It is not recommended to use a bidirectional ObjectServer Gateways to connect two distinct IBM Netcool/OMNIbus systems due to the race conditions that invariably occur as a result. Similarly, it is not recommended to use two unidirectional Gateways to provide bidirectional flow between two distinct IBM Netcool/OMNIbus system due to both the possibility of race conditions, and the potential to end up with looping events (ie. since two unidirectional ObjectServer Gateways have no awareness of the other's state and can potentially send events back and forth endlessly). Anyone implementing such a configuration does so at their own risk.

The best practice Gateway configuration for normal ObjectServer Gateways within a multi-ObjectServer environment can be found in the standard multitier architecture configuration in `$OMNIHOME/extensions/multitier/gateway/`. See the section entitled *A starting point* in *Chapter 2 Planning* of this document for more information on how to apply the standard multitier architecture configuration.

Note: Many of the concepts introduced in this chapter are contained within the standard multitier architecture configuration.

Generic properties

Although the *IBM Netcool/OMNIbus 8.1 Probe & Gateway Guide* contains a full description of all Gateway generic properties, a number of key properties are highlighted in this section and should be set according to the advice given.

MaxLogFileSize

The default maximum log file size for Gateways is 1024 (KB). It is recommended to increase this value up to 2048 (KB) for production systems to allow for the logging of more information.

MessageLevel

The default message level for Gateways is "warn", which causes the Gateway to log all *fatal*, *error*, *warn* messages. It is recommended to increase the logging level to "info", which causes the Gateway to log all *fatal*, *error*, *warn*, and *info* messages. This provides an additional level of information in the Gateway logs, which helps Netcool Administrators monitor Gateway activity more effectively.

Gate.CacheHashTblSize

The Gateway stores information about the events it is transferring in an internal cache, which is implemented in software as a hash table. This property defines the size of the hash table used in the Gateway cache.

More than one data item (ie. table row) can be stored in each hash table entry, however performance may be degraded if the number of data items per hash table entry is too many. On the other hand, having less than one data item per hash table entry (ie. on average) means the Gateway's memory footprint will be unnecessarily large.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

As a general guideline therefore, it is recommended to set the `Gate.CacheHashTblSize` property to the nearest prime number that is at least *half* the maximum estimated number of events or rows in the `alerts.status` table. This means each hash table entry will have on average two data items stored within it.

For Collection to Aggregation ObjectServer Gateways however, it is recommended to set the `Gate.CacheHashTblSize` property to the nearest prime number that is at least *one quarter* the maximum estimated number of events or rows in the `alerts.status` table. This means each hash table entry will have on average four data items stored within it.

A vanilla ObjectServer Gateway has the `Gate.CacheHashTblSize` value set to 5023, priming it for a maximum size of around 10,000 standing rows in the `alerts.status` table under typical use.

The standard multitier architecture configuration has the `Gate.CacheHashTblSize` value set to 10069 for a Collection to Aggregation Gateway, priming it to be able to handle a maximum size of around 40,000 standing rows in the `alerts.status` table. This property is set to 50021 for the Aggregation failover Gateway and Aggregation to Display Gateways, priming them to be able to handle a maximum size of around 100,000 standing rows in the `alerts.status` table.

Note: Since the Collection layer ObjectServers only store approximately the last 10 minutes' worth of data (ie. event data is reaped approximately 10 minutes after it is forwarded to the Aggregation layer), the Collection to Aggregation Gateways' `Gate.CacheHashTblSize` value does not need to be as large as for the other Gateways. See *Chapter 7 Anatomy of the standard multitier architecture configuration* on page 129 for more information on the setting of this property within the context of the standard multitier architecture configuration.

dumpprops property

To inspect a Gateway's current properties settings, simply run the Gateway with the `dumpprops` switch as in the following example:

```
$OMNIHOME/bin/nco_g_objserv_uni -dumpprops
```

Note: The output will consider any properties set in the default properties file.

File locations

When IBM Netcool/OMNIbus is installed, the generic ObjectServer Gateway configuration files are installed into the `$OMNIHOME/gates/` directory, one subdirectory for the configuration files for unidirectional Gateways and one subdirectory for those of bidirectional Gateways.

The best practice multitier architecture ObjectServer Gateway configuration files are in the `$OMNIHOME/extensions/multitier/gateway/` directory.

When any other Gateway patches are installed, their respective default Gateway configuration files are installed in `$OMNIHOME/gates/` under a subdirectory for each Gateway (eg. `remedy/`, `db2/`, `oracle/`, etc.).

It is a best practice to never modify any of the above default files. If any of the files are required for use, they must be first copied to the `$OMNIHOME/etc/` directory and the copy modified instead. This is because the original files need to be retained as a baseline point of reference. Additionally, when subsequent product updates are released in the form of patches, service packs or interim fixes, any of the files in the default locations may be overwritten by the update.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Original files and backups

After making a copy of default Gateway configuration files into the `$OMNIHOME/etc/` directory, they will be customised to suit the environment. If later updates are subsequently required, the current version of the file should be backed up before any modifications are made, so that the update can be easily rolled back, if required.

Just as for Probe configuration files, a suggested naming convention for backed up Gateway configuration files is to append the copied filename with a `.<date>` file extension. eg:

```
$ cp AGG_GATE.map AGG_GATE.map.20110923
```

Gateway configuration files

This section contains best practice guidelines for the configuration of Gateway configuration files.

The purpose of the Gateway properties file is to set the various properties of the Gateway.

The purpose of the Gateway mapping file is to map fields in the source ObjectServer to the target fields. Gateway replication file entries require and refer to Gateway mapping file entries.

The purpose of the Gateway table replication file is to specify which tables to replicate between the source ObjectServer and the target and what sort of changes to transfer (ie. INSERTS, UPDATES, DELETES).

Gateway mapping files

When adding custom fields to existing mappings (eg. the `StatusMap` that defines mappings for the `alerts.status` table), it is a best practice to place the custom fields at the bottom of the mapping, after all the default fields. Secondly, it is a best practice to clearly demarcate the custom additions from the default field mappings by using comments.

The standard multitier architecture Gateway mapping files provide a demarcated area within the `StatusMap` mapping section for the addition of custom fields. It is used as per the following example:

```
##################################################################
#
#  CUSTOM alerts.status FIELD MAPPINGS GO HERE
#
##################################################################

   'MyCustomField'      =  '@MyCustomField',

##################################################################
```

EXAMPLE:

Widgetcom have a custom table they need replicated between their primary and backup Aggregation ObjectServers. The Netcool Administrator appends the following section of code to the end of the

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

bidirectional Aggregation Gateway mapping file to provide this functionality, and includes the Gateway's mapping file in the solution delivery package:

```
#####################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 26-Sep-2011
# The following table mapping has been added to enable the
# replication of the Widgetcom custom table custom.locations.
#####################################################################

CREATE MAPPING LocationsMap
(
   'LocationID'            = '@LocationID',
   'LocationName'          = '@LocationName',
   'LocationCity'          = '@LocationCity',
   'LocationCountry'       = '@LocationCountry'
);

#####################################################################
```

NOTES:

- The mapping definition name follows the naming convention of the other tables in the mapping file in terms of naming and letter case usage.
- The mapping definition has a commented header including who modified the file last and when they did it, plus a description of what the mapping is for. It also includes a footer line.
- All lines are tabbed to ensure alignment in the file. Ensure that whitespace usage and layout within the mapping file is consistent and aids readability.
- The replication file definition for this example is included in the next section.

Gateway table replication files

When adding custom entries to existing Gateway replication definition files, it is a best practice to place these entries at the bottom of the file, after the default entries. Secondly, it is a best practice to clearly demarcate the custom additions from the default entries by using comments.

EXAMPLE:

Widgetcom have a custom table they need replicated between their primary and backup Aggregation ObjectServers. They have already defined the mapping in the mapping file, now they need to define the replication behaviour by creating a replication file entry. The Netcool Administrator appends the following section of code to the end of the bidirectional Aggregation Gateway table replication file to provide this functionality, and includes the Gateway's table replication file in the solution delivery package:

```
#####################################################################
# Last modified by: J Smith (jsmith) - Netcool Admin - 26-Sep-2011
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
# The following table replication definition has been added to
# enable the replication of the Widgetcom custom table
# custom.locations using the table mapping LocationsMap.
###################################################################

REPLICATE ALL FROM TABLE 'custom.locations'
   USING MAP 'LocationsMap'
   INTO 'custom.locations';

###################################################################
```

NOTES:

- The replication definition file entry has a commented header including who modified the file last and when they did it, plus a description of what the entry is for. It also includes a footer line.
- All lines are tabbed to ensure alignment in the file. Ensure that whitespace usage and layout within the mapping file is consistent and aids readability.

Cold standby Gateways (UNIX only)

There are occasions where one needs to set up a cold standby Netcool Gateway — for example: a cold standby ticketing Gateway or a DB2 archiving database Gateway.

Included in Appendix B is a UNIX shell script (`nco_test_gateway.sh`) that will provide cold standby Gateway management functionality. An IBM Netcool Administrator should review the script before use, and modify the environment parameters to suit. It should be configured to run under Process Agent control (not PA aware) on the host machine where the backup Gateway will run.

The script works by using the `nco_ping` utility to monitor the "primary" Gateway. If the script fails an `nco_ping` attempt more than `RETRIES` number of times (a modifiable parameter in the script), then the script makes a call to `nco_pa_start` to start up the cold standby Gateway. After this point, the script will continue to try to `nco_ping` the primary Gateway. Once the script eventually receives a positive response from the primary Gateway, the script makes a call to `nco_pa_stop` to stop the standby Gateway, and returns to its original state of monitoring the primary Gateway's availability.

This script was originally written to provide functionality to enable and disable triggers on the backup ObjectServer in a failover pair, however it became redundant when this functionality was built in to the ObjectServer via Gateway signals and new triggers introduced into IBM Netcool/OMNIbus. The script was subsequently adapted (and renamed) to provide the functionality described above instead.

NOTES:

- A username (who is a member of the `ncoadmin` UNIX group if local file authentication is in use) and password needs to be included in the script to enable the script to make calls to the Process Agent. For this reason, this script should have its file permissions modified to be readable only by the specified user. For example:

  ```
  chmod 711 nco_test_gateway.sh
  ```

- This script is provided "as-is" as a solution for cold standby Gateway functionality and is intended for use on UNIX-based systems only.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServer Gateway resynchronisation types

The resynchronisation type (specified by the property `Gate.Resync.Type`) specifies *how* a Gateway carries out resynchronisation between two ObjectServers when the Gateway either starts up or restores a lost connection.

As previously mentioned in this document, it is a best practice to use the standard multitier architecture configuration whenever deploying more than one ObjectServer.

On occasion however, there are scenarios where a custom configuration may be required — for example: when events need to be forwarded to another IBM Netcool/OMNIbus installation, during an upgrade migration. For this reason, this section provides some notes about the three different ObjectServer Gateway resynchronisation types: NORMAL, UPDATE, MINIMAL, and TWOWAYUPDATE — along with some usage guidance.

When a Gateway resynchronises, the *resynchronisation "master"* is defined as the ObjectServer from which the data will be copied or read from, and the *resynchronisation "slave"* is defined as the ObjectServer where the source data will be written to.

For unidirectional Gateways, the direction of resynchronisation is only ever in one direction: from the `Gate.Reader.Server` (master) to the `Gate.Writer.Server` (slave). In the case of a bidirectional ObjectServer Gateway, the direction of resynchronisation is determined by the property `Gate.Resync.Master` or `Gate.Resync.Preferred` if both ObjectServers have been up the same amount of time (ie. uptime). In all other cases, the master for the resynchronisation process will be the ObjectServer that has the greatest uptime.

The terms "master" and "slave" will be referred to within the ObjectServer Gateway resynchronisation type descriptions that follow.

NORMAL resynchronisation type

When an ObjectServer Gateway has the `Gate.Resync.Type` property set to NORMAL, the Gateway will perform the resynchronisation in the following manner:

- For each of the tables referenced in the Gateway table replication file (`*.tblrep.def`), the Gateway will delete all data from the slave ObjectServer, and then transfer a full copy from the master ObjectServer to the slave.

After resynchronisation has completed, the master and slave ObjectServers will be synchronised.

Before selecting this resynchronisation type for a custom solution, the following points should be considered:

- Any table rows that exist on the slave that do not exist on the master will be lost.
- Any table rows that exist on both master and slave ObjectServers will result in the copy of the row from the master being retained on both. Any prior updates to the slave version of the data will be lost.

UPDATE resynchronisation type

When an ObjectServer Gateway has the `Gate.Resync.Type` property set to UPDATE, the Gateway will perform the resynchronisation in the following manner:

- For each of the tables referenced in the Gateway table replication file (`*.tblrep.def`), the Gateway will build up a cache containing all rows in each of the master and slave ObjectServers. The Gateway will then examine the cache contents for each table and compare the row data from the master with the row data from the slave.
- Any rows in the slave ObjectServer that also exist in the master will get updated with the master ObjectServer's copy, if they are different.
- Any rows that are in the master but not in the slave will be copied to the slave.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- Any rows in the slave that are not in the master will be untouched.

After resynchronisation has completed, the slave ObjectServer will have a copy of every event in the master ObjectServer. The slave however will also *retain* the events it has that the master does not have.

Before selecting this resynchronisation type for a custom solution, the following points should be considered:

- No events will be lost (ie. this resynchronisation type is lossless).
- After resynchronisation has completed, the contents of the master and slave ObjectServers may not be the same.

Note: UPDATE is the resynchronisation type used by the Collection to Aggregation Gateways in the standard multitier architecture configuration due to its lossless nature. See *Chapter 7 Anatomy of the standard multitier architecture configuration* in this document for more information on the resynchronisation types used and the design rationale behind it.

MINIMAL resynchronisation type

Note: This resynchronisation type has been superseded by: TWOWAYUPDATE.

When an ObjectServer Gateway has the `Gate.Resync.Type` property set to MINIMAL, the Gateway will perform the resynchronisation in the following manner:

- For each of the tables referenced in the Gateway table replication file (`*.tblrep.def`), the Gateway will build up a cache containing all rows in each of the master and slave ObjectServers. The Gateway will then examine the cache contents for each table and compare the row data from the master with the row data from the slave.
- Any rows that are in the master but not in the slave will be copied to the slave.

For the `alerts.status` table, there are a few more steps that are taken:

- The Gateway will update all rows on the slave ObjectServer setting `OldRow` to 1, which effectively marks them for deletion.
- The Gateway will determine the set of rows that are in both master and slave ObjectServers, and set `OldRow` to 0 for those rows in the slave ObjectServer, which is thereby unmarking them for deletion.
- The Gateway will get the `LastUpdate` time from both master and slave ObjectServers.
- The Gateway will select all rows on the master ObjectServer that have a `StateChange` value more recent than the `LastUpdate` time on the slave ObjectServer, and then send those rows to the slave.
- A default trigger running on the slave ObjectServer called `pass_deletes` will delete any rows in the `alerts.status` table where `OldRow = 1`.

Note: The default trigger `pass_deletes` is disabled by default and must be enabled if this functionality is required.

This resynchronisation type is essentially the same as UPDATE, however MINIMAL simply has the additional step of marking for deletion events (ie. rows in the `alerts.status` table) in the slave that are not in the master by setting the field `OldRow` to 1 for those events in the slave.

After resynchronisation has completed (and the trigger `pass_deletes` has fired on the slave), the master and slave ObjectServers will be synchronised.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Before selecting this resynchronisation type for a custom solution, the following points should be considered:

- Any table rows that exist on the slave that do not exist on the master will be lost.
- Any table rows that exist on both master and slave ObjectServers will result in the copy of the row from the master being retained on both, if the `StateChange` of the master event is newer than the `LastUpdate` time on the slave. Any prior updates to the slave version of the row will potentially be lost therefore.

The benefit of this resynchronisation type over NORMAL, is that MINIMAL replicates the behaviour of a NORMAL resynchronisation, without deleting all the rows first and then recopying them over. Hence this method *minimises* the amount of data to be re-sent when resynchronisation takes place. Note however that if the master and the slave ObjectServer datasets are significantly out of synch with each other, it limits to a degree the efficiency gains this resynchronisation type results in.

MINIMAL resynchronisation type is particularly of value when archiving Gateways (ie. such as DB2 or Oracle) are connected to the slave ObjectServer. A MINIMAL Gateway resynchronisation type means the archiving Gateway won't send false deletes and then subsequently new inserts of the same events. This is what happens during a NORMAL resynchronisation.

Note: Because MINIMAL resynchronisation mode touches all events on the slave ObjectServer when it sets `OldRow` to 1, it creates a lot of IDUC traffic for any IDUC clients connected to the slave ObjectServer. It is not necessarily a problem but something to bear in-mind if bandwidth is an issue between the slave ObjectServer and the clients.

TWOWAYUPDATE resynchronisation type

When an ObjectServer Gateway has the `Gate.Resync.Type` property set to TWOWAYUPDATE, the Gateway will perform the resynchronisation in the following manner. Initially, the Gateway will follow the same steps as an UPDATE resynchronisation:

- For each of the tables referenced in the Gateway table replication file (`*.tblrep.def`), the Gateway will build up a cache containing all rows in each of the master and slave ObjectServers. The Gateway will then examine the cache contents for each table and compare the row data from the master with the row data from the slave.
- Any rows in the slave ObjectServer that also exist in the master will get updated with the master ObjectServer's copy, if they are different.
- Any rows that are in the master but not in the slave will be copied to the slave.
- Any rows in the slave that are not in the master will be untouched.

Then the following additional steps are followed:

- The event caches of the master and slave ObjectServers are compared again.
- Any rows that are in the slave but not in the master are written to the master.

After the TWOWAYUPDATE resynchronisation is complete, the master and slave contain identical rows. This achieves the same outcome as setting a bidirectional ObjectServer Gateway to MINIMAL however in a much more efficient way, without requiring additional ObjectServer automation in order to make it work properly.

Note: TWOWAYUPDATE is the resynchronisation type used by the Aggregation failover Gateways in the standard multitier architecture configuration due to its lossless nature. See *Chapter 7 Anatomy of the standard multitier architecture configuration* in this document for more information on the resynchronisation types used and the design rationale behind it.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServer Gateway resynchronisation lock types

The Gateway resynchronisation lock type (specified by the property `Gate.Resync.LockType`) specifies the ObjectServer *locking* mechanism that will be used when a Gateway carries out resynchronisation between two ObjectServers. A Gateway resynchronisation occurs every time a Gateway either starts up or restores a lost connection. When a Gateway "locks" an ObjectServer during a resynchronisation, all other processes in the ObjectServer are placed on hold until the Gateway has completed its synchronisation and removes the lock.

As previously mentioned in this document, it is a best practice to use the standard multitier ObjectServer configuration whenever deploying more than one ObjectServer.

On occasion however, there are scenarios where a custom configuration may be required — for example: when events need to be forwarded to another IBM Netcool/OMNIbus installation. For this reason, this section provides some notes about the three different ObjectServer Gateway resynchronisation lock types: FULL, PARTIAL, and NONE — along with some usage guidance.

FULL resynchronisation lock type

When an ObjectServer Gateway has the `Gate.Resync.LockType` property set to FULL, the Gateway will lock *both* the master *and* the slave ObjectServers for the duration of the synchronisation process.

Before selecting this resynchronisation lock type for a custom solution, the following points should be considered:

- *Both* master and slave ObjectServers will be exclusively locked by the Gateway for the duration of the resynchronisation process and *neither* will be able to fulfil any requests from connected clients or internal processes for the duration of the resynchronisation process.
- *No* client connections to the master or the slave will be permitted during the resynchronisation and any previously connected client may time out as a result.
- This lock type guarantees that the resynchronisation process will be allowed to complete before any data is modified on either the master or slave.

PARTIAL resynchronisation lock type

When an ObjectServer Gateway has the `Gate.Resync.LockType` property set to PARTIAL, the Gateway will *only* lock the slave ObjectServer for the duration of the synchronisation process.

Before selecting this resynchronisation lock type for a custom solution, the following points should be considered:

- Although the slave ObjectServer will be exclusively locked by the Gateway for the duration of the resynchronisation process, the master ObjectServer will still be available to process requests by connected clients or internal processes.
- *No* client connections to the slave will be permitted during the resynchronisation and any previously connected client may time out as a result.
- This lock type guarantees that the resynchronisation process will be allowed to complete before any data is modified on the slave. Any updates to events that occur on the master while the resynchronisation is in progress will be passed to the slave during the next IDUC cycle.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Note: PARTIAL is the resynchronisation lock type used by the standard multitier configuration bidirectional failover ObjectServer Gateway AGG_GATE. This setting locks the slave ObjectServer during a resynchronisation to prevent components failing over or failing back between the primary and backup until the correct time. Failback of components from backup to primary is strictly staged and controlled to ensure a smooth transition. See *Chapter 7 Anatomy of the standard multitier architecture configuration* for more information on the resynchronisation lock types used and the design rationale behind it.

NONE resynchronisation lock type

When an ObjectServer Gateway has the `Gate.Resync.LockType` property set to NONE, the Gateway will lock *neither* the master *nor* the slave ObjectServer for the duration of the synchronisation process.

Before selecting this resynchronisation lock type for a custom solution, the following points should be considered:

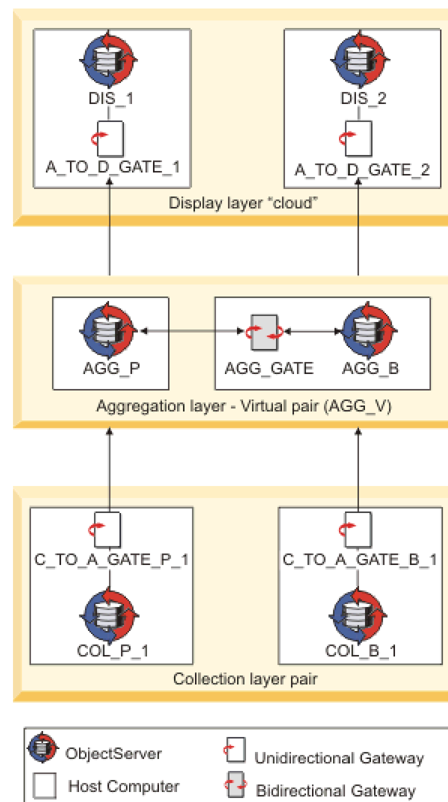
- *Neither* the master nor the slave ObjectServer will be locked by the Gateway for the duration of the resynchronisation process and *both* will be able to fulfil any requests from connected clients or internal processes for the duration of the resynchronisation process.
- Client connections to both the master and the slave are permitted during the resynchronisation.
- This lock type does not guarantee that data will not be modified on either the master or slave before the resynchronisation process completes. Any updates to any rows missed by the synchronisation process will however propagate across the Gateway during the next IDUC cycle.

Note: NONE is the resynchronisation lock type used by the standard multitier configuration inter-tier ObjectServer Gateways, that is, Collection to Aggregation and Aggregation to Display Gateways. This setting allows ObjectServers to continue to respond to connected clients, even though a resynchronisation is in progress. It is not necessary to lock either master or slave ObjectServers when inter-tier Gateways are resynchronising. See *Chapter 7 Anatomy of the standard multitier architecture configuration* in this document for more information on the resynchronisation lock types used and the design rationale behind it.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 7 Anatomy of the standard multitier architecture configuration

The standard multitier architecture configuration is the best practice pre-canned configuration for building 1, 2, or 3-tiered IBM Netcool/OMNIbus environments. It is a best practice to use the standard multitier architecture configuration as a baseline configuration when deploying an IBM Netcool/OMNIbus system where more than one ObjectServers are deployed and connected.



This chapter provides a brief anatomy of the standard multitier configuration including notes relating to the design rationale.

Note: For more information on *how* to deploy the standard multitier architecture configuration, see the *Chapter 8: Configuring and deploying a multitiered architecture* in the *IBM Netcool/OMNIbus 8.1 Installation & Deployment Guide*.

This chapter begins with a description of how failback work differently between the Collection and Aggregation layers; then lists the file locations of the standard multitier configuration; and finally follows with a detailed description of the Collection layer, Aggregation layer and Display layer component configurations. It should be reiterated here that both the Collection and Display layers are *optional*, and their inclusion in an architecture design is dependent on system loading or other requirements. For further reading regarding when to consider deploying Collection or Display ObjectServers, see *Chapter 2: Planning* in this document.

Failback at Collection and Aggregation

The Aggregation layer primary and backup ObjectServers contain the authoritative copy of the event set and are synchronised via a bidirectional failover Gateway. A primary/backup pair of ObjectServers at the Collection layer, however, are not connected in any way. Hence the needs of an Aggregation

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

ObjectServer pair with respect to failing clients over and back are different to those of a Collection ObjectServer pair.

If there is an outage of a primary Aggregation ObjectServer which subsequently is restored, the primary and backup Aggregation ObjectServers must resynchronise with each other before the system can return to normal operation.

If there is an outage of a primary Collection ObjectServer which subsequently is restored, there is no resynchronisation process that needs to occur between it and its counterpart backup Collection ObjectServer. The primary and backup members of a Collection ObjectServer pair will invariably contain different event sets under normal conditions.

Consequently, the way failback has been designed to work at the Aggregation layer in the standard multitier configuration is different to the way failback has been designed to work at the Collection layer. This section attempts to explain the differences between the two mechanisms, primarily around how failback is orchestrated, and the reasons for the differences.

Failback at the Aggregation layer

Since resynchronisation between the primary and backup Aggregation ObjectServers is an essential part of resuming normal service, the failback mechanism has deliberately been designed to be staged and managed via automated processes within the backup Aggregation ObjectServer, called *controlled failback*. For this reason, the automatic failback mechanisms of Probes and Gateways connecting to the Aggregation layer are purposely disabled so that each component doesn't individually try to initiate failback on its own.

Under the *controlled failback* mechanism, the bidirectional failover Gateway AGG_GATE that connects the primary and backup Aggregation ObjectServers is first allowed time to reconnect to both Aggregation ObjectServers and perform its resynchronisation step before any of the connected client processes are disconnected from the backup Aggregation ObjectServer and naturally fail back to the primary Aggregation ObjectServer.

The process for how failback has been designed to work at the Aggregation layer is summarised by the following:

1. Primary Aggregation ObjectServer goes down; all connected processes fail over to the backup Aggregation ObjectServer.
2. The primary Aggregation ObjectServer is restored.
3. The bidirectional failover Gateway AGG_GATE reconnects to both the primary and the backup Aggregation ObjectServers and carries out a resynchronisation.

Note: The bidirectional failover Gateway AGG_GATE has its *Resync.LockType* set to PARTIAL which means the target ObjectServer, in this case the primary Aggregation ObjectServer, is locked for the duration of the resynchronisation step. The target ObjectServer will not allow any connections to it during this time until the resynchronisation completes and the lock is released. The source ObjectServer, in this case the backup Aggregation ObjectServer, is *not* locked during the resynchronisation step, however, and so connected clients can continue to function as normal.

---

4. When the bidirectional failover Gateway AGG_GATE completes its resynchronisation step, it raises a "resynchronisation complete" signal in both Aggregation ObjectServers.
5. A signal trigger in the backup Aggregation ObjectServer called: *disconnect_all_clients* detects this signal. The trigger sets the *ActingPrimary* property to FALSE and then disconnects all connected client processes except for Administrator sessions, Webtop/WebGUI sessions or connections established by the bidirectional failover Gateway: AGG_GATE.
6. All clients detect the disconnection and attempt to reconnect in the normal way by first attempting to connect to the primary Aggregation ObjectServer, which is now available again.
7. All processes successfully reconnect to the primary Aggregation ObjectServer and normal service resumes.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

NOTES:

- The failback is controlled, and all clients fail back at the same time rather than independently of each other, and only fail back after the resynchronisation step has completed.
- If it is the backup Aggregation ObjectServer that goes down, the process is the same as above and there is no impact to any other process, since there should be no clients connected to the backup Aggregation ObjectServer under normal operation.
- Any Gateways connected to the Aggregation layer should have the failback property set to FALSE in the properties file to ensure failback is not done autonomously. Note that this is pre-configured to be this way in the standard multitier configuration.
- Any Probes connected to the Aggregation layer should have the *PollServer* property set to the default of 0 (ie: zero), which effectively disables automatic failback. See the section entitled *Generic properties* in *Chapter 5: Probes* for more details on the recommended settings for generic Probe properties.

Failback at the Collection layer

Failback at the Collection layer is much simpler than at the Aggregation layer since there is no resynchronisation process *between* the primary and backup Collection ObjectServers.

Note: Counterpart primary and backup Collection ObjectServers are not connected to each other in any way and have no awareness of each other's state, nor do they need to. They run completely independently of each other.

FAILOVER AND FAILBACK OF PROBES

Failover of a Probe connected to a primary Collection ObjectServer will occur if the primary Collection ObjectServer goes down or if the Probe loses its connection to the primary Collection ObjectServer for whatever reason.

Once connected to the backup Collection ObjectServer, the Probe will periodically poll the primary Collection ObjectServer to see if it has been restored. If the primary Collection ObjectServer has been restored, the Probe will automatically fail back.

This polling and failback behaviour is activated in Probes by configuring the following:

- The *BackupObjectServer* property being set to TRUE in the backup Collection ObjectServer (done automatically in the standard multitier configuration);
- The *PollServer* property being set to a non-zero value in the Probe's properties file.

Note: A value of 120 (seconds) is suggested for the *PollServer* property. See the section entitled *Generic properties* in *Chapter 5: Probes* for more details on the recommended settings for generic Probe properties.

FAILOVER AND FAILBACK OF OTHER PROCESSES

Netcool Gateways connecting to a Collection ObjectServer pair will fail over and fail back between the primary and backup by default, assuming the Gateway's local interfaces file is configured correctly. Failover and failback are default behaviour and must be explicitly disabled by the setting of a property in the Gateway's properties file.

The standard multitier configuration explicitly disables this behaviour for inter-tier Collection to Aggregation Gateway Reader connections, since each Collection ObjectServer has its own dedicated Collection to Aggregation Gateway. These settings are all pre-configured in the standard multitier configuration.

For all other Gateways — for example: JDBC Gateways connecting to the Collection layer for archiving purposes — automatic failback should be enabled in the normal way via the correct

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

definition of the primary and backup Collection ObjectServers within the Gateway's local interfaces file.

Note: Since automatic failback is the default behaviour in Gateways, nothing special must be done to enable this behaviour in terms of setting Gateway properties.

Automatic failover and failback can also be configured within IBM Netcool/Impact datasources and should be done so whenever connecting IBM Netcool/Impact to a Collection layer pair of ObjectServers — for example: for the purposes of one-time enrichment of events.

Any other processes connecting to a Collection ObjectServer pair — for example: third-party scripts — should be aware of the primary/backup setup and should fail over and back in a similar manner as other Netcool processes.

File locations

The standard multitier architecture configuration ships with IBM Netcool/OMNIbus 8.1 and is included in the `$OMNIHOME/extensions/multitier/` directory.

Under this directory are two further subdirectories:

- `objectserver/`

  This directory contains SQL files required to convert a vanilla ObjectServer into either a Collection, an Aggregation, or a Display ObjectServer. Also included in this directory are "roll-back" SQL files used to undo the modifications made by the SQL files.

- `gateway/`

  This directory contains all the necessary Gateway configuration files for use within a standard multitier environment.

Note: See *Chapter 7. Configuring and deploying a multitiered architecture* in the *IBM Netcool/OMNIbus 8.1 Installation & Deployment Guide* for detailed instructions on how to use these files when deploying a standard multitier architecture.

The Collection layer

This section contains design notes on configuration of the Collection layer components.

Collection layer ObjectServer triggers

This section contains design notes on the Collection layer ObjectServer components contained within the file:

`$OMNIHOME/extensions/multitier/objectserver/collection.sql`

- A number of ObjectServer fields are created within the `alerts.status` table for use at the Collection layer:

Field name	Type	Purpose
SentToAgg	Integer	This field is used as a flag to control when the Collection to Aggregation Gateway should forward events to the Aggregation layer. This is toggled by ObjectServer triggers and the Collection to Aggregation Gateway.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

CollectionExpireTime	Timestamp	This field is used to set how long an event should persist at the Collection layer before it is reaped. Note that no events will be reaped until they have been forwarded to the Aggregation layer first. A default value of 30 seconds is applied to this field for every event if it is not already set within the Probe rules. This allows time for events to deduplicate between batches of events being sent up from the Collection layer to the Aggregation layer.
CollectionFirst AggregationFirst DisplayFirst	Timestamp	These fields are used by the TimeToDisplay functionality. See the section entitled *The TimeToDisplay field* later in this chapter for an explanation of how these fields are used.

- The trigger `generic_clear` is disabled at the Collection layer. Problem events that occurred more than a few minutes ago are unlikely to still be on the Collection layer ObjectServer since they are reaped within 4 minutes after they have been forwarded up to the Aggregation layer. This time amount is based on: the 30 second CollectionExpireTime of the event, the 31 second timing of the `col_expire` trigger, the 120 seconds that a cleared event should persist until it is deleted, and the 60 second timing of the `delete_clears` trigger.

  If a resolution event subsequently arrives at the Collection layer, it should be forwarded to the Aggregation layer as-is. If it were processed (and cleared) by the `generic_clear` trigger at the Collection layer first, it would not subsequently correlate against the existing problem event at the Aggregation layer when it arrives as it would likely already be cleared; meaning it would not be processed by the `generic_clear` trigger at the Aggregation layer.

- The default insert trigger `new_row` is disabled and replaced by a priority 2 database insert trigger called `col_new_row`. This trigger handles the initial setting of fields for new inserts.

  The length of time an event will persist at the Collection layer is 30 seconds by default, assuming it has been forwarded to the Aggregation layer. This is defined via the field `CollectionExpireTime`. Note that this field can be pre-set in the Probe rules. This trigger only sets the `CollectionExpireTime` if it is not already set at the time it is inserted into the Collection ObjectServer.

- The default trigger `deduplication` is disabled and replaced by a priority 2 reinsert database trigger called `col_deduplication`.

  This trigger handles the discarding of duplicate events from Probes when they are replayed on failover or failback, as well as the updating of the key fields on deduplication.

- The default trigger `expire` is disabled and replaced by a priority 2 temporal trigger called `col_expire`. This trigger handles the removal of expired events at the Collection layer. This replacement is necessary over the default expiry trigger because it only expires events that have both been sent up the Aggregation layer already and where the event has not been modified for longer than `CollectionExpireTime` seconds ago.

- The trigger `timestamp_inserts` is a priority 3 insert database trigger created to set the field `CollectionFirst` to the current timestamp. This field is used by the *TimeToDisplay* functionality. See the section entitled *The TimeToDisplay field* later in this chapter for more information on these fields and their purpose.

- The trigger `resend_events_on_failover` is a priority 1 signal trigger that is set to fire when the Collection to Aggregation ObjectServer Gateway has finished resynchronisation. A Gateway resynchronisation will happen when a Gateway initially starts, fails over, or fails back. When the Gateway resynchronises, it only sends event data that satisfies the Gateway filter — namely: `'SentToAgg = 0'`.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Under certain Gateway failure scenarios however, there is a possibility that events picked up by the Gateway (and hence flagged as having been sent by the AFTER IDUC DO directive) did not actually make it to the Aggregation ObjectServer. This trigger therefore simply resends all events presently in the Collection ObjectServer, to ensure that the Aggregation ObjectServer receives at least one copy of each event.

At the time this trigger fires, the Gateway has already completed its resynchronisation process. The affected events will therefore all be forwarded to the Aggregation ObjectServer in the next IDUC cycle.

- Next, the relevant permissions are created and applied to the newly created triggers.
- A procedure called `configure_backup_objectserver` is created in the ObjectServer to initially configure the ObjectServer as a backup, where appropriate. This procedure is then executed once and then dropped from the ObjectServer.
- Finally, accelerated event channels are set up to propagate events through the tiers immediately as they arrive. This significantly reduces the amount of time it takes for events to propagate through a tiered environment and be visible to operators.

The Collection Gateway properties file

This section contains design notes relating to the Collection layer Gateway properties files named `C_TO_A_GATE_P_1.props` and `C_TO_A_GATE_B_1.props`. Below are listed the properties included in the file `C_TO_A_GATE_P_1.props` as a reference.

- `MaxLogFileSize                : 2048`

  This property defines the maximum log file size is increased to 2048 (kb) from the default of 1024 to allow for more logging information to be captured.

- `MessageLevel                  : 'info'`

  This property defines the logging message level is raised from 'warn' to 'info' to allow for informational logging information to be captured.

- `MessageLog              : '$OMNIHOME/log/C_TO_A_GATE_P_1.log'`

  This property defines the name of the log file the Gateway will use.

- `Name                          : 'C_TO_A_GATE_P_1'`

  This property defines the name of the Gateway as per the interfaces file.

- `Ipc.Timeout                   : 90`

  This property defines the number of seconds the Gateway will wait before it times out while waiting for a response from the ObjectServers. This overrides the TCP/IP timeout which can be up to 10 minutes, depending on the settings of the host.

  In the scenario of a network cable disconnect, the Gateway may not be able to detect the disconnection, and so may only fail over or attempt to reconnect after the TCP/IP timeout. This property ensures the Gateway will respond in a timelier fashion. A value of 90 seconds has been selected for the standard multitier architecture configuration as it has been deemed a good balance between being too impatient, and taking too long to respond to a disconnect.

  In environments with high numbers of events, high event churn or due to other performance degrading factors, it is possible that a single SQL command can take longer than 90 seconds to complete. When this happens, the Gateway will assume it has lost its connection and attempt to reconnect. This can cause the Gateway to get into an endless loop of connecting, timing out and reconnecting. Although this scenario is rare in practice, it can happen. In such cases, it is necessary to increase the *Ipc.Timeout* value. The amount that the property value should be increased will vary from one environment to the next but should be increased to a value that includes a degree of contingency to avoid it happening again.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- `Gate.CacheHashTblSize         : 10069`

  The Gateway uses a hash table cache to store details of tables that require transferring from one ObjectServer to another. The main function of the cache is to facilitate journal and details table insert operations. When a journal or detail is forwarded for insertion into a target ObjectServer, the Gateway writer needs to know the corresponding status `Serial` in the target ObjectServer. This information is found in the cache. The cache is also used for any other tables specified using the table replication definition table.

  The cache aids performance optimisation by providing the Gateway with an in-memory summarised view of the contents of the ObjectServers to which it is linked. This means that the Gateway does not have to query an ObjectServer to check for the existence of an event, or the `Serial` value or `Tally` value of an event; it can simply check the cache of the target ObjectServer instead.

  By default, the size of the hash table cache is 5023 elements (or rows). For maximum performance, the value of the hash table cache size would be set to a number that is similar or greater than the number of rows in the `alerts.status` table — however it should be noted that increasing the size of the hash table cache also increases the Gateway's memory requirement.

  The guidance for Collection to Aggregation ObjectServer Gateways is to set this value to the nearest prime number that is no less than a quarter of the maximum number of rows expected in the Collection layer ObjectServer. For example, the standard multitier architecture uses a value of 10069 for Collection to Aggregation Gateways — which is therefore suitable for a Collection ObjectServer holding up to around 40,000 events in the `alerts.status` table. If the Collection layer was going to hold up to 60,000 events in the `alerts.status` table, for example, then a recommended minimum value for this property would be 15013.

Note: To maximize efficiency, you should specify a prime number for this property.

- `Gate.MapFile                  : '$OMNIHOME/etc/C_TO_A_GATE.map'`

  This property defines the name of the mapping file the Gateway will use. Note that Collection Gateway mapping files tend to be the same.

- `Gate.Mapper.ForwardHistoricDetails  : TRUE`

  One of the replication settings for Collection to Aggregation Gateways is that all updates are converted to inserts. The property `Gate.Mapper.ForwardHistoricDetails` specifies whether or not all details for each event will be forwarded when an update is converted to an insert by the Gateway.

  This is set to TRUE for the standard multitier architecture because an UPDATE of an event on the Collection ObjectServer that gets sent up to the Aggregation layer is usually a new occurrence of an event, or where the Gateway is resynchronising. This property ensures that all details belonging to events being forwarded by the Gateway make it up to the Aggregation layer under the various failover/failback scenarios. Note that the Aggregation layer is configured to discard the receipt of duplicate details, and so it is not a problem if details are inadvertently forwarded twice.

- `Gate.Mapper.ForwardHistoricJournals : TRUE`

  One of the replication settings for Collection to Aggregation Gateways is that all updates are converted to inserts. The property `Gate.Mapper.ForwardHistoricJournals` specifies whether or not all journals for each event will be forwarded when an update is converted to an insert by the Gateway.

  This is set to TRUE for the standard multitier architecture because an UPDATE of an event on the Collection ObjectServer that gets sent up to the Aggregation layer is usually a new occurrence of an event, or where the Gateway is resynchronising. This property ensures that all journals belonging to events being forwarded by the Gateway make it up to the Aggregation layer under the various failover/failback scenarios. Note that the Aggregation layer is configured to discard

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

the receipt of duplicate journals, and so it is not a problem if journals are inadvertently forwarded twice.

- `Gate.StartupCmdFile: '$OMNIHOME/gates/objserv_uni/objserv_uni.startup.cmd'`

  This property defines the name of the start-up command file the Gateway will use. Since the standard multitier architecture start-up command file does not contain any commands and is identical to the default, this property is simply set to read the default start-up command file on start-up. A duplicate of the default file for each instance of the Gateway is not required therefore, since it is not modified in any way.

- `Gate.Reader.Server : 'COL_P_1'`

  This property specifies the name of the ObjectServer the Gateway will read from. This example is set to read from the primary Collection layer ObjectServer COL_P_1.

- `Gate.Reader.Description : 'collection_gate'`

  This property defines the "description" the Gateway will pass to the Collection layer ObjectServer it is reading from. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.Reader.FailbackEnabled : FALSE`

  Since each Collection ObjectServer has its own dedicated Gateway connecting it to the Aggregation layer, the Gateway Reader that connects to the Collection ObjectServer requires neither failover nor failback. This property is set to disable failback behaviour therefore.

- `Gate.Reader.IducFlushRate : 30`

  If this property is not set, the Gateway will only collect IDUC when it is prompted to do so by the Collection ObjectServer it is reading from. With this property set, the Gateway will also initiate its own IDUC collections every 30 seconds, in addition to the ObjectServer prompted IDUC collections. This property has been set to reduce the time it takes for events to propagate through the tiered environment.

- `Gate.Reader.TblReplicateDefFile: '$OMNIHOME/etc/C_TO_A_GATE_P_1.tblrep.def'`

  This property defines the name of the table replication file the Gateway will use. The table replication file defines which tables are replicated between the source and target ObjectServers, and the options that are to be applied when replication is done. See the next section for more details on the contents of this file.

- `Gate.Reader.IgnoreStatusFilter : TRUE`

  This property was created for Gateways where a filter is in use, as is the case for Collection to Aggregation ObjectServer Gateways.

  During the IDUC process, three of the tables that are typically replicated are the `alerts.status`, `alerts.journal` and `alerts.details` tables. For efficiency, the Gateway will automatically not send journals or details where the corresponding event (ie. from the `alerts.status` table) does not match the Gateway's filter.

  For a Collection Gateway in the standard multitier architecture, after the Gateway has forwarded event data from the event table, it executes SQL on the Collection ObjectServer to flag those events as having been sent (sets `SentToAgg = 1`). This flag intentionally causes these events to be blocked by the filter during the next successive IDUC cycle. This functionality is by design so that event forwarding to the Aggregation layer is strictly controlled. If an event recurs or the Gateway resynchronises, triggers reset the flag, which allows the events to be re-forwarded by the Gateway again.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Since the Gateway gathers IDUC data each table in turn however, by the time it considers the rows to be transferred from the journal and details table, the corresponding events from the event table have already been processed by the Gateway, and had the flag set. This would normally have the result of blocking any journals and details for these events from being forwarded to the Aggregation layer. This property however specifies that when the Gateway is considering rows to be forwarded from the journal and details tables, it should ignore this filter with respect to the event table, and allow the rows from the journal and details tables to be sent anyway. Without this property set, no journals or details would be propagated from Collection to Aggregation.

- `Gate.Writer.Server                 : 'AGG_V'`

  This property specifies the name of the ObjectServer the Gateway will write to. The value of this property will be the same for all Collection to Aggregation ObjectServer Gateways, and is set to the virtual Aggregation layer ObjectServer pair `AGG_V`.

- `Gate.Writer.Description            : 'collection_gate'`

  This property defines the "description" the Gateway will pass to the Aggregation layer ObjectServer it is writing to. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.Writer.BufferSize             : 50`

  When the Gateway is transferring data from one ObjectServer to the other, it batches SQL statements together. This property specifies the number of SQL statements the Gateway should buffer or group together from the source ObjectServer before sending to the destination ObjectServer. It has been determined, under lab conditions, that a value of 50 for the batch size gives optimal transferral rates for a typical Gateway.

- `Gate.Writer.FailbackEnabled        : FALSE`

  The WRITER part of the Gateway is the part that connects to the Aggregation ObjectServer layer. Since the timing of failback of Gateways connected to the Aggregation layer is controlled by the backup Aggregation ObjectServer (by design), the automatic failback feature of connecting Gateways should be disabled. This property (set to FALSE) disables the automatic failback feature of the WRITER part of Collection to Aggregation Gateways.

- `Gate.Writer.SAF                    : TRUE`

  The Gateway can store data it has read from the Collection ObjectServer if it loses connection to the Aggregation layer. On successful reconnection, it will replay these events from the store and resume normal operation. This property is set to TRUE within the standard multitier architecture configuration.

- `Gate.Writer.SAFFile:`
  `'$OMNIHOME/var/objserv_uni/C_TO_A_GATE_P_1.store'`

  This property specifies the name of the file that will be used to store data the Gateway has read from the Collection ObjectServer if it enters store-and-forward mode. (See the previous property for more information on store-and-forward mode.)

- `Gate.Writer.UseBulkInsCmd          : TRUE`

  This property (when set to TRUE, as in this case) causes the Gateway to optimise the format of SQL insert commands into batches before sending them to the destination Aggregation ObjectServer. This enables the Aggregation ObjectServer to process the insert commands more efficiently.

- `Gate.Resync.Enable                 : TRUE`

  This property ensures the Gateway resynchronisation functionality is enabled.

- `Gate.Resync.Type                   : 'UPDATE'`

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This property specifies that the Gateway resynchronisation type is set to UPDATE. This ensures that rows that exist on the Aggregation ObjectServer that don't exist on the Collection ObjectServer aren't inadvertently deleted during the resynchronisation process.

- `Gate.Resync.LockType : 'NONE'`

  This property specifies that the Gateway should not attempt to gain a lock on either the source or destination ObjectServers during the resynchronisation process. This setting allows all inter-tier Gateways to resynchronise simultaneously rather than one at a time.

The Collection Gateway table replication file

This section contains design notes on the Collection layer Gateway table replication files named `C_TO_A_GATE_P_1.tblrep.def` and `C_TO_A_GATE_B_1.tblrep.def`. The following notes use the contents of the `C_TO_A_GATE_P_1.tblrep.def` as an example.

With respect to the status table, the table replication file for the Collection Gateway contains the following:

```
REPLICATE INSERTS, UPDATES
        FROM TABLE 'alerts.status'
        USING MAP 'StatusMap'
        FILTER WITH 'SentToAgg = 0'
        ORDER BY 'Serial ASC'
        SET UPDTOINS CHECK TO FORCED
        AFTER IDUC DO 'Tally = 0, SentToAgg = 1'
        CACHE FILTER 'ServerName = \'COL_P_1\'';
```

NOTES:

- `REPLICATE INSERTS, UPDATES`

  Only INSERTS and UPDATES are replicated from the Collection layer to the Aggregation layer for the `alerts.status` table. This is because data from this table is reaped from the Collection layer. If the Gateway replicated deletes, the rows from this table would also be deleted incorrectly at the Aggregation layer. Note that this principle should be considered for any custom tables that are created at the Collection layer and propagated to the Aggregation layer.

- `FILTER WITH 'SentToAgg = 0'`

  The Gateway has a FILTER applied so that it only forward events where the flag field `SentToAgg = 0`. After the Gateway sends each event up to the Aggregation layer, the AFTER IDUC DO directive sets the `SentToAgg` field on each sent event to 1, hence preventing it from being included in any further IDUC transfers. This field is configured to be reset to 0 if another occurrence of the event occurs (ie. a deduplication) or if the Gateway resynchronises (courtesy of the Collection ObjectServer trigger named `resend_events_on_failover`).

- `ORDER BY 'Serial ASC'`

  The Gateway has an ORDER BY directive applied that will cause the events to be transferred up to the Aggregation layer in the order that they were received, in ascending order of the `Serial` field.

- `SET UPDTOINS CHECK TO FORCED`

  Under standard operation, the Gateway maintains a cache of the events on both the source and target ObjectServers. On processing ObjectServer IDUC, the Gateway differentiates an insert

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

from an update by whether the event to be transferred from the source ObjectServer exists on the destination ObjectServer. If it does not exist on the destination ObjectServer, it sends the event as an INSERT. If it does exist on the destination ObjectServer, it sends the event as an UPDATE.

The SET UPDTOINS CHECK TO FORCED directive tells the Gateway not to bother checking its cache when processing updates, and simply to send all updates as inserts. Hence the Gateway will not differentiate between an insert and an update, and simply send the events for both categories as inserts.

By not having to check its cache, this speeds up the transfer of `alerts.status` IDUC data. It also simplifies the handling of the event rows at the destination ObjectServer. An incoming insert at the Aggregation layer will be treated as such if the event does not already exist and get processed by the insert trigger: `agg_new_row`. If the event already exists, the insert will be handled by the reinsert trigger: `agg_deduplication` — which has specifically been designed for this functionality. Due to this property setting, there is no need to handle incoming updates from the Collection layer.

- `AFTER IDUC DO 'Tally = 0, SentToAgg = 1'`

  After the Gateway reader has read the events from the Collection layer ObjectServer, the AFTER IDUC DO directive is executed against those rows.

  When an Aggregation ObjectServer receives a row from the Collection layer, it is ambiguous as to whether the `Tally` field holds the number of newly inserted rows since the last IDUC cycle, or whether it reflects an increment to the previous `Tally`. To manage this ambiguity, the Gateway simply sends the current value of `Tally` up to the Aggregation layer, and then zeroes the Tally at the Collection layer via the AFTER IDUC DO directive. This means the reinsert trigger at the Aggregation layer only needs to add the incoming value from the Collection layer to the running total it holds on deduplication, thereby resolving the ambiguity.

  The setting of `SentToAgg = 1` is discussed above, and provides a mechanism to control when events are picked up for forwarding by the Gateway.

- `CACHE FILTER 'ServerName = \'COL_P_1\'';`

  Prior to a resynchronisation, the Gateway will read the events on both the source and destination ObjectServers, so it can determine which events will be transferred during the resynchronisation step. The destination ObjectServer (ie. the Aggregation layer) may contain events from many Collection ObjectServers, however the only events that are relevant to the Gateway with respect to a resynchronisation are events that have originated from the Collection ObjectServer this Gateway is connected to as its source.

  The CACHE FILTER directive causes the Gateway to apply a filter when building its cache so that it limits the selected events to only those originating from its own Collection layer ObjectServer. The filter is applied on both the source and destination SELECT queries when the cache is constructed, and is tied to the Collection ObjectServer's `ServerName`.

  On building the initial resynchronisation cache, when the Gateway reads from the source Collection ObjectServer, it will automatically include all events (since all events in the Collection ObjectServer will have that `ServerName`), however when the Gateway reads from the destination Aggregation ObjectServer, it will include just the subset of events that have originated from this Collection ObjectServer. This often results in a substantially smaller resynchronisation cache, and speeds up resynchronisation time as a result.

With respect to the journal table, the table replication file for the Collection Gateway contains the following:

```
REPLICATE INSERTS, UPDATES FROM TABLE 'alerts.journal'
   USING MAP 'JournalMap';
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

NOTES:

- Just as for the status table, only INSERTS and UPDATES of journals are replicated to the Aggregation layer. This is because when the journal rows are reaped from the Collection layer, they should not be removed at the Aggregation layer. The deletion of journal data from the Aggregation layer is managed locally within the Aggregation layer.

With respect to the details table, the table replication file for the Collection Gateway contains the following:

```
REPLICATE INSERTS, UPDATES FROM TABLE 'alerts.details'
   USING MAP 'DetailsMap';
```

NOTES:

- Just as for the status table, only INSERTS and UPDATES of details are replicated to the Aggregation layer. This is because when the detail rows are reaped from the Collection layer, they should not be removed at the Aggregation layer. The deletion of detail data from the Aggregation layer is managed locally within the Aggregation layer.

Note: If any custom tables need to be replicated from the Collection layer to the Aggregation layer, replication definitions should be created and appended to the end of this file. Note that a corresponding mapping must also exist in the Gateway's mapping file. See the section entitled *Gateway configuration files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway replication file entries and mappings.

The Collection Gateway mapping file

The Collection Gateway mapping file (named `C_TO_A_GATE.map`) provides the following section to insert custom field mappings for the `alerts.status` table:

```
#####################################################################
#
#  CUSTOM alerts.status FIELD MAPPINGS GO HERE
#
#####################################################################




#####################################################################
```

Note: Any custom table mappings should be appended to the end of the mapping file. See the section entitled *Gateway mapping files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway mappings.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Aggregation layer

This section contains design notes on configuration of the Aggregation layer components.

Aggregation layer ObjectServer triggers

This section contains design notes on the Aggregation layer ObjectServer components contained within the file:

`$OMNIHOME/extensions/multitier/objectserver/aggregation.sql`

- A number of ObjectServer fields are created within the `alerts.status` table for use at the Aggregation layer:

Field name	Type	Purpose
CollectionFirst	Timestamp	These fields are used by the TimeToDisplay functionality. See the section entitled The TimeToDisplay field later in this chapter for an explanation of how these fields are used.
AggregationFirst		
DisplayFirst		

- The Native Event List load-balancing table `master.servergroups` is reset. There is an option included after this line to uncomment rows to populate the table with load-balancing control data. If more than two Display ObjectServers are to be included in the load-balancing, additional lines can be added at this point.
- The default insert trigger `new_row` is disabled and replaced by a priority 2 database insert trigger called `agg_new_row`. This trigger handles the initial setting of fields for new inserts.

  If the incoming event is not coming from a Gateway, then the `ServerName` and `ServerSerial` are initialised with local values, since the event is then likely being inserted by a Probe.

  The `FirstOccurrence`, `LastOccurrence`, and `Tally` fields are initialised if the incoming event has no values set for these fields.
- The default trigger `deduplication` is disabled and replaced by a priority 2 reinsert database trigger called `agg_deduplication`.

  This trigger handles the discarding of duplicate events from Probes when they are replayed on failover or failback.

  If the incoming reinsert is coming from the failover Gateway, the update action is simply to replace the existing event with the incoming event. If not, then the following updates are applied.

  If the incoming reinsert is coming from a Collection to Aggregation Gateway, the incoming `Tally` value is added to the existing `Tally` value, since the incoming value represents the number of recurrences of this event that have occurred since the last occurrence of this event.

  If the reinsert is not coming from either a Collection or Aggregation Gateway, then it is likely coming from a Probe, therefore the `Tally` field is simply incremented by 1.

  If the incoming reinsert `LastOccurrence` value is set to a non-zero value, then update the `LastOccurrence` with the incoming value, else update it with the current time.

  The core fields are also updated on deduplication, including the `Severity` field. There is an option here to modify the behaviour of severity updates on deduplication. See the SQL file for details.

  Finally, `StateChange` and `InternalLast` are updated with the current time on deduplication.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- The database trigger `timestamp_inserts` is a priority 3 insert database trigger created to set the field `AggregationFirst` to the current timestamp. This field is used by the *TimeToDisplay* functionality. See the section entitled *The TimeToDisplay field* later in this chapter for more information on these fields and their purpose.
- The failback triggers are created to control the behaviour of the Aggregation ObjectServers during a failback scenario to control the sequence of operations.
- The timing of the failback of clients connected to the backup Aggregation ObjectServer is controlled by the backup Aggregation ObjectServer. All Probes and Gateways connecting to the Aggregation layer should have failback disabled so that the timing of their disconnection can be controlled centrally.

  A signal trigger named `disconnect_all_clients` is created on both Aggregation ObjectServers, but only enabled on the backup. On detecting when the failover bidirectional ObjectServer Gateway AGG_GATE finishes its resynchronisation, the trigger disconnects all connected clients; with the exception of Netcool/Administrator, any IBM Netcool/Webtop or IBM Netcool/OMNIbus WebGUI connections and the bidirectional Gateway. All other clients — including Gateways, Probes, and IBM Netcool/Impact — are disconnected however, and will each independently attempt to reconnect to the virtual Aggregation ObjectServer pair. Since the primary Aggregation ObjectServer is now up and finished resynchronisation, the disconnected clients will be able to reconnect to the primary.

Note: Since all inter-tier Gateways are configured not to lock on resynchronisation, they can all resynchronise simultaneously.

- As each Gateway completes its resynchronisation, it will raise a `gw_resync_finish` signal in each ObjectServer it is connected to. The trigger `resync_complete` is a priority 1 signal trigger created in Aggregation ObjectServers to generate synthetic events that notify that a Gateway has completed its resynchronisation. A *"resynchronisation complete"* synthetic event will be generated for each connected Gateway.

  The trigger is in the `primary_only` trigger group and is therefore only enabled on one of the Aggregation ObjectServers at a time. These synthetic events provide useful information to an IBM Netcool Administrator during a failover or a failback to notify them that Gateways have resynchronised successfully.
- The relevant permissions are created and applied to the newly created triggers.
- A conversion is created for the TimeToDisplay functionality. See the section entitled *The TimeToDisplay field* later in this chapter for more information.
- A procedure called `set_up_objectservers` is created to configure the ObjectServer as a backup where appropriate and also to enable or disable certain triggers depending on whether or not it is a primary or a backup Aggregation ObjectServer. This procedure is executed once and then dropped.
- Finally, accelerated event channels are set up to propagate events through the tiers immediately as they arrive. This significantly reduces the amount of time it takes for events to propagate through a tiered environment and be visible to operators.

The Aggregation failover Gateway properties file

This section contains design notes relating to the Aggregation layer failover Gateway properties files named `AGG_GATE.props`.

- `MaxLogFileSize                : 16384`

  This property defines the maximum log file size is increased to 16384 (kb) from the default of 1024 to allow for more logging information to be captured.
- `MessageLevel                  : 'info'`

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This property defines the logging message level is raised from “warn” to “info” to allow for informational logging information to be captured.

- `MessageLog                : '$OMNIHOME/log/AGG_GATE.log'`

  This property defines the name of the log file the Gateway will use.

- `Name                      : 'AGG_GATE'`

  This property defines the name of the Gateway as per the interfaces file.

- `Ipc.Timeout               : 90`

  This property defines the number of seconds the Gateway will wait before it times out while waiting for a response from the ObjectServers. This overrides the TCP/IP timeout which can be up to 10 minutes, depending on the settings of the host.

  In the scenario of a network cable disconnect, the Gateway may not be able to detect the disconnection, and so may only fail over or attempt to reconnect after the TCP/IP timeout. This property ensures the Gateway will respond in a timelier fashion. A value of 90 seconds has been selected for the standard multitier architecture configuration as it has been deemed a good balance between being too impatient and taking too long to respond to a disconnect.

  In environments with high numbers of events, high event churn or due to other performance degrading factors, it is possible that a single SQL command can take longer than 90 seconds to complete. When this happens, the Gateway will assume it has lost its connection and attempt to reconnect. This can cause the Gateway to get into an endless loop of connecting, timing out and reconnecting. Although this scenario is rare in practice, it can happen. In such cases, it is necessary to increase the *Ipc.Timeout* value. The amount that the property value should be increased will vary from one environment to the next but should be increased to a value that includes a degree of contingency to avoid it happening again.

- `Gate.CacheHashTblSize     : 50021`

  The Gateway uses a hash table cache to store details of tables that require transferring from one ObjectServer to another. The main function of the cache is to facilitate journal and details table insert operations. When a journal or detail is forwarded for insertion into a target ObjectServer, the Gateway writer needs to know the corresponding status `Serial` in the target ObjectServer. This information is found in the cache. The cache is also used for any other tables specified using the table replication definition table.

  The cache aids performance optimisation by providing the Gateway with an in-memory summarised view of the contents of the ObjectServers to which it is linked. This means that the Gateway does not have to query an ObjectServer to check for the existence of an event, or the `Serial` value, or `Tally` value of an event; it can simply check the cache of the target ObjectServer instead.

  By default, the size of the hash table cache is 5023 elements (or rows). For maximum performance, the value of the hash table cache size would be set to a number that is similar or greater than the number of rows in the `alerts.status` table, however it should be noted that increasing the size of the hash table cache also increases the Gateway’s memory requirement.

  The guidance for the Aggregation layer bidirectional Gateway is to set this value to the nearest prime number that is no less than half the maximum number of rows expected in the Aggregation ObjectServers. Note that this provides for a bigger cache than for Collection to Aggregation Gateways. This additional allowance for the Aggregation layer bidirectional Gateway is due to the heavier load this type of Gateway invariably supports due to the typically higher event numbers it transfers.

  The standard multitier architecture uses a value of 50021 for the bidirectional failover Aggregation Gateway, which is therefore suitable for an Aggregation ObjectServer holding up to around 100,000 events in the `alerts.status` table.

Note: To maximize efficiency, you should specify a prime number for this property.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- `Gate.MapFile : '$OMNIHOME/etc/AGG_GATE.map'`

  This property defines the name of the mapping file the Gateway will use.

- `Gate.StartupCmdFile: '$OMNIHOME/gates/objserv_bi/objserv_bi.startup.cmd'`

  This property defines the name of the start-up command file the Gateway will use. Since the standard multitier architecture start-up command file does not contain any commands and is identical to the default, this property is simply set to read the default start-up command file on start-up.

- `Gate.Mapper.ForwardHistoricDetails : TRUE`

  An update will be converted to an insert by the Gateway if an update from the source ObjectServer is of an event that event does not exist on the destination ObjectServer — ie. there is no event to update on the destination ObjectServer. The property `Gate.Mapper.ForwardHistoricDetails` specifies whether or not all details for the converted event will also be forwarded when this happens. This is set to TRUE for the bidirectional failover Gateway as this is desired behaviour in all circumstances.

- `Gate.Mapper.ForwardHistoricJournals : TRUE`

  An update will be converted to an insert by the Gateway if an update from the source ObjectServer is of an event that event does not exist on the destination ObjectServer — ie. there is no event to update on the destination ObjectServer. The property `Gate.Mapper.ForwardHistoricJournals` specifies whether or not all journals for the converted event will also be forwarded when this happens. This is set to TRUE for the bidirectional failover Gateway as this is desired behaviour in all circumstances.

- `Gate.ObjectServerA.Server : 'AGG_P'`

  This property specifies the primary Aggregation ObjectServer the Gateway will connect to.

- `Gate.ObjectServerA.BufferSize : 50`

  When the Gateway is transferring data from one ObjectServer to the other, it batches SQL statements together. This property specifies the number of SQL statements the Gateway should buffer or group together from the source ObjectServer before sending to the destination ObjectServer. It has been determined, under lab conditions, that a value of 50 for the batch size gives optimal transferral rates for a typical Gateway.

- `Gate.ObjectServerA.Description : 'failover_gate'`

  This property defines the "description" the Gateway will pass to the primary Aggregation ObjectServer it is connected to. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.ObjectServerA.TblReplicateDefFile: '$OMNIHOME/etc/AGG_GATE.tblrep.def'`

  This property defines the name of the table replication file the Gateway will use for replicating from the primary Aggregation ObjectServer to the backup. The table replication file defines which tables are replicated between the source and target ObjectServers, and the options that are to be applied when replication is done. See the next section for more details on the contents of this file.

- `Gate.ObjectServerA.UseBulkInsCmd : TRUE`

  This property (when set to TRUE, as in this case) causes the Gateway to optimise the format of SQL insert commands into batches before sending them to the counterpart Aggregation ObjectServer. This enables the counterpart Aggregation ObjectServer to process the insert commands more efficiently.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- `Gate.ObjectServerB.Server          : 'AGG_B'`

  This property specifies the backup Aggregation ObjectServer the Gateway will connect to.

- `Gate.ObjectServerB.BufferSize      : 50`

  When the Gateway is transferring data from one ObjectServer to the other, it batches SQL statements together. This property specifies the number of SQL statements the Gateway should buffer or group together from the source ObjectServer before sending to the destination ObjectServer. It has been determined, under lab conditions, that a value of 50 for the batch size gives optimal transferral rates for a typical Gateway.

- `Gate.ObjectServerB.Description     : 'failover_gate'`

  This property defines the "description" the Gateway will pass to the backup Aggregation ObjectServer it is connected to. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.ObjectServerB.TblReplicateDefFile: '$OMNIHOME/etc/AGG_GATE.tblrep.def'`

  This property defines the name of the table replication file the Gateway will use for replicating from the primary Aggregation ObjectServer to the backup. The table replication file defines which tables are replicated between the source and target ObjectServers, and the options that are to be applied when replication is done. See the next section for more details on the contents of this file.

- `Gate.ObjectServerB.UseBulkInsCmd   : TRUE`

  This property (when set to TRUE, as in this case) causes the Gateway to optimise the format of SQL insert commands into batches before sending them to the counterpart Aggregation ObjectServer. This enables the counterpart Aggregation ObjectServer to process the insert commands more efficiently.

- `Gate.Resync.Enable                 : TRUE`

  This property ensures the Gateway resynchronisation functionality is enabled.

- `Gate.Resync.Type                   : 'TWOWAYUPDATE'`

  This property specifies that the Gateway resynchronisation type is set to TWOWAYUPDATE. This ensures that the superset of the events on both ObjectServers are retained during the resynchronisation process. Where the same event exists on both ObjectServers, the copy of the event on the master ObjectServer will overwrite the copy of the event on the slave.

- `Gate.Resync.LockType               : 'PARTIAL'`

  This property specifies that the Gateway should only attempt to gain a lock on the destination ObjectServer during the resynchronisation process. This setting allows the restored Aggregation ObjectServer time to resynchronise with the master before clients begin connecting to it and resynchronising with it. It also means that the master ObjectServer for the resynchronisation process is not locked during the resynchronisation, and is therefore available to service connected clients in the meantime. This is an important feature and ensures continuation of service from the Aggregation layer during a failover or a failback scenario.

Note: The fail back of connected clients is strictly controlled between the Aggregation ObjectServer pair. See the notes about the failback triggers in the previous section of this document entitled: *Aggregation layer ObjectServer triggers*.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Aggregation failover Gateway table replication file

This section contains notes on the Aggregation layer Gateway table replication file named `AGG_GATE.tblrep.def`. With respect to the status, journal and details tables, the table replication file contains the following:

```
REPLICATE ALL FROM TABLE 'alerts.status'
   USING MAP 'StatusMap';

REPLICATE ALL FROM TABLE 'alerts.journal'
   USING MAP 'JournalMap';

REPLICATE ALL FROM TABLE 'alerts.details'
   USING MAP 'DetailsMap';
```

This is default, vanilla settings for the replication of these tables.

In contrast to the default table replication file however, all other table replications are uncommented for use including: the user related system tables, the desktop related system tables and the `master.servergroups` table (used for automatic load-balancing of Native Event List clients).

Note: If any custom tables need to be replicated between the Aggregation ObjectServer pair, replication definitions should be created and appended to the end of this file. Note that a corresponding mapping must also exist in the Gateway's mapping file. See the section entitled: *Gateway configuration files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway replication file entries and mappings.

The Aggregation failover Gateway mapping file

The Aggregation Gateway mapping file (named `AGG_GATE.map`) provides the following section to insert custom field mappings for the `alerts.status` table:

```
##################################################################
#
#  CUSTOM alerts.status FIELD MAPPINGS GO HERE
#
##################################################################




##################################################################
```

Note: Any custom table mappings should be appended to the end of the mapping file. See the section entitled *Gateway mapping files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway mappings.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Display layer

This section contains design notes on configuration of the Display layer components.

Display layer ObjectServer triggers

This section contains design notes on the Display layer ObjectServer components contained within the file:

`$OMNIHOME/extensions/multitier/objectserver/display.sql`

- A number of ObjectServer fields are created within the `alerts.status` table for use at the Display layer:

Field name	Type	Purpose
CollectionFirst	Timestamp	These fields are used by the TimeToDisplay functionality. See the section entitled The TimeToDisplay field later in this chapter for an explanation of how these fields are used.
AggregationFirst		
DisplayFirst		
TimeToDisplay	Integer	This field is used to store the number of seconds it has taken for each event to arrive at the Display layer ObjectServer from its point of insertion into the multitier environment. The point of insertion may be either the Collection or the Aggregation layer. See the triggers contained in this section for further information about how the TimeToDisplay value is calculated.

- The `master.national` table is cleared and populated with a row for the virtual Aggregation ObjectServer name `AGG_V`.
- The default insert trigger `new_row` is disabled and replaced by a priority 2 insert database trigger called `dsd_new_row`. This trigger handles the initial setting of certain fields for new inserts.

  If an insert is being done by anything other than the Aggregation to Display Gateway and by anyone other than the root user, the insert is cancelled.

  If the insert is not being done by the Aggregation to Display Gateway, the `ServerName` and `ServerSerial` are initialised.

Note: The only event that should be inserted at the Display layer other than by the Aggregation to Display Gateway is the synthetic `TimeToDisplay` event. This event is inserted by a trigger and hence `%user.user_id` will appear as the root user (ie. 0).

- The default trigger `deduplication` is disabled and replaced by a priority 2 reinsert database trigger called `dsd_deduplication`.

  This trigger first stores the current values of two TimeToDisplay variables as well as the `ServerName` and `ServerSerial` into local variables.

  If the reinsert is not coming from either the Aggregation to Display Gateway or being done by the root user, the reinsert is cancelled.

  The existing row is then replaced in its entirety by the incoming row.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Next, the values stored in local variables are restored back to the original fields so that they retain their original values prior to the deduplication. This is done so that the synthetic TimeToDisplay events do not lose their settings for these fields.

In all cases, the `InternalLast` and `StateChange` fields are updated with the current timestamp.

- The next five triggers provide the performance statistics functionality that indicates the time to display each event as well as a synthetic event generated every minute that shows an average *TimeToDisplay* value for all events currently in the Display ObjectServer.
- The trigger `timestamp_inserts` is a priority 3 insert database trigger created to set the field `DisplayFirst` to the current timestamp and then calculate the `TimeToDisplay` value for the current event. See the section entitled *The TimeToDisplay field* later in this chapter for more information on these fields and their purpose.
- The trigger `tag_old_events` is a priority 1 signal trigger that fires the moment the Aggregation to Display Gateway completes its resynchronisation process, and sets the `TimeToDisplay` field of all events currently in the Display ObjectServer to 9999999. This is effectively a means of "tagging" these events so that they can be excluded from any subsequent average `TimeToDisplay` calculations.

  The reason this is necessary is because the `DisplayFirst` value is reset every time the Gateway resynchronises, but the fields `CollectionFirst` and `AggregationFirst` retain their original values, which may be very old in comparison. This has the effect of artificially skewing the `TimeToDisplay` values upwards, hence these events are necessarily are excluded from the average calculations as a result.
- The trigger `calculate_time_to_display` is a priority 3 temporal trigger that tallies up the valid `TimeToDisplay` values of all events currently in the Display ObjectServer and then works out an average. The trigger then creates/updates a synthetic event unique to the local Display ObjectServer showing the average time to display all events in the ObjectServer.
- The triggers `disable_average_calculation` and `enable_average_calculation` are priority 1 signal triggers that disable and enable the average `TimeToDisplay` calculation trigger when the Aggregation to Display Gateway respectively starts and finishes its resynchronisation. The purpose of this to prevent misleading averages being presented to the users if the calculation is made part way through the resynchronisation process.
- Next, the relevant permissions are created and applied to the newly created triggers.
- Finally, the trigger group `primary_only` is disabled. The triggers in this group should not be enabled on a Display layer ObjectServer.

The Display Gateway properties file

This section contains design notes relating to the Display layer Gateway properties files named `A_TO_D_GATE_1.props` and `A_TO_D_GATE_2.props`. Below are listed the properties included in the file `A_TO_D_GATE_1.props` as a reference.

- `MaxLogFileSize                    : 2048`

  This property defines the maximum log file size is increased to 2048 (kb) from the default of 1024 to allow for more logging information to be captured.
- `MessageLevel                      : 'info'`

  This property defines the logging message level is raised from "warn" to "info" to allow for informational logging information to be captured.
- `MessageLog                   : '$OMNIHOME/log/A_TO_D_GATE_1.log'`

  This property defines the name of the log file the Gateway will use.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- `Name                          : 'A_TO_D_GATE_1'`

  This property defines the name of the Gateway as per the interfaces file.

- `Ipc.Timeout                   : 90`

  This property defines the number of seconds the Gateway will wait before it times out while waiting for a response from the ObjectServers. This overrides the TCP/IP timeout which can be up to 10 minutes, depending on the settings of the host.

  In the scenario of a network cable disconnect, the Gateway may not be able to detect the disconnection, and so may only fail over or attempt to reconnect after the TCP/IP timeout. This property ensures the Gateway will respond in a timelier fashion. A value of 90 seconds has been selected for the standard multitier architecture configuration as it has been deemed a good balance between being too impatient and taking too long to respond to a disconnect.

  In environments with high numbers of events, high event churn or due to other performance degrading factors, it is possible that a single SQL command can take longer than 90 seconds to complete. When this happens, the Gateway will assume it has lost its connection and attempt to reconnect. This can cause the Gateway to get into an endless loop of connecting, timing out and reconnecting. Although this scenario is rare in practice, it can happen. In such cases, it is necessary to increase the *Ipc.Timeout* value. The amount that the property value should be increased will vary from one environment to the next but should be increased to a value that includes a degree of contingency to avoid it happening again.

- `Gate.CacheHashTblSize         : 50021`

  The Gateway uses a hash table cache to store details of tables that require transferring from one ObjectServer to another. The main function of the cache is to facilitate journal and details table insert operations. When a journal or detail is forwarded for insertion into a target ObjectServer, the Gateway writer needs to know the corresponding status `Serial` in the target ObjectServer. This information is found in the cache. The cache is also used for any other tables specified using the table replication definition table.

  The cache aids performance optimisation by providing the Gateway with an in-memory summarised view of the contents of the ObjectServers to which it is linked. This means that the Gateway does not have to query an ObjectServer to check for the existence of an event, or the `Serial` value or `Tally` value of an event; it can simply check the cache of the target ObjectServer instead.

  By default, the size of the hash table cache is 5023 elements (or rows). For maximum performance, the value of the hash table cache size would be set to a number that is similar or greater than the number of rows in the `alerts.status` table, however it should be noted that increasing the size of the hash table cache also increases the Gateway's memory requirement.

  The guidance for the Display layer Gateway is to set this value to the nearest prime number that is no less than half the maximum number of rows expected in the Aggregation ObjectServers. Note that this provides for a bigger cache than for Collection to Aggregation Gateways. This additional allowance for the Aggregation and Display Gateways is due to the heavier load these types of Gateways invariably support due to the typically higher event numbers they transfer.

  The standard multitier architecture uses a value of 50021 for the Display Gateways, which is therefore suitable for an Aggregation/Display ObjectServer holding up to around 100,000 events in the `alerts.status` table.

Note: To maximize efficiency, you should specify a prime number for this property.

- `Gate.MapFile                  : '$OMNIHOME/etc/A_TO_D_GATE.map'`

  This property defines the name of the mapping file the Gateway will use. Note that Display Gateway mapping files tend to be the same.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

- `Gate.StartupCmdFile: '$OMNIHOME/gates/objserv_uni/objserv_uni.startup.cmd'`

  This property defines the name of the start-up command file the Gateway will use. Since the standard multitier architecture start-up command file does not contain any commands and is identical to the default, this property is simply set to read the default start-up command file on start-up. A duplicate of the default file for each instance of the Gateway is not required therefore, since it is not modified in any way.

- `Gate.Mapper.ForwardHistoricDetails : TRUE`

  One of the replication settings for Aggregation to Display Gateways is that all updates are converted to inserts. The property `Gate.Mapper.ForwardHistoricDetails` specifies whether or not all details for each event will be forwarded when an update is converted to an insert by the Gateway.

  This property ensures that all details belonging to events being forwarded by the Gateway make it up to the Display layer under the various failover/failback scenarios.

- `Gate.Mapper.ForwardHistoricJournals : TRUE`

  One of the replication settings for Collection to Aggregation Gateways is that all updates are converted to inserts. The property `Gate.Mapper.ForwardHistoricJournals` specifies whether or not all journals for each event will be forwarded when an update is converted to an insert by the Gateway.

  This property ensures that all journals belonging to events being forwarded by the Gateway make it up to the Display layer under the various failover/failback scenarios.

- `Gate.Reader.Server : 'AGG_V'`

  This property specifies the name of the ObjectServer the Gateway will read from. The value of this property will be the same for all Aggregation to Display ObjectServer Gateways, and is set to the virtual Aggregation layer ObjectServer pair `AGG_V`.

- `Gate.Reader.Description : 'display_gate'`

  This property defines the "description" the Gateway will pass to the Aggregation layer ObjectServer it is reading from. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.Reader.FailbackEnabled : FALSE`

  The READER part of the Gateway is the part that connects to and reads from the Aggregation ObjectServer layer. Since the timing of failback of Gateways connected to the Aggregation layer is controlled by the backup Aggregation ObjectServer (by design), the automatic failback feature of connecting Gateways should be disabled. This property (set to FALSE) disables the automatic failback feature of the READER part of Collection to Aggregation Gateways.

- `Gate.Reader.IducFlushRate : 30`

  If this property is not set, the Gateway will only collect IDUC when it is prompted to do so by the Aggregation ObjectServer it is reading from. With this property set, the Gateway will also initiate its own IDUC collections every 30 seconds, in addition to the ObjectServer prompted IDUC collections. This property has been set to reduce the time it takes for events to propagate through the tiered environment.

- `Gate.Reader.TblReplicateDefFile: '$OMNIHOME/etc/A_TO_D_GATE.tblrep.def'`

  This property defines the name of the table replication file the Gateway will use. The table replication file defines which tables are replicated between the source and target ObjectServers,

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

and the options that are to be applied when replication is done. See the next section for more details on the contents of this file.

- `Gate.Writer.Description : 'display_gate'`

  This property defines the "description" the Gateway will pass to the Display layer ObjectServer it is writing to. This value is accessible from within the ObjectServer by examining the variable `%user.description`. See the standard multitier architecture ObjectServer SQL files in `$OMNIHOME/extensions/multitier/objectserver/` for examples of how this variable is accessed and used.

- `Gate.Writer.Server : 'DIS_1'`

  This property specifies the name of the ObjectServer the Gateway will write to. This example is set to read from the Display layer ObjectServer `DIS_1`.

- `Gate.Writer.BufferSize : 50`

  When the Gateway is transferring data from one ObjectServer to the other, it batches SQL statements together. This property specifies the number of SQL statements the Gateway should buffer or group together from the source ObjectServer before sending to the destination ObjectServer. It has been determined, under lab conditions, that a value of 50 for the batch size gives optimal transferral rates for a typical Gateway.

- `Gate.Writer.SAF : TRUE`

  The Gateway can store data it has read from the Aggregation layer if it loses connection to the Display ObjectServer. On successful reconnection, it will replay these events from the store and resume normal operation. This property is set to TRUE within the standard multitier architecture configuration.

- `Gate.Writer.SAFFile: '$OMNIHOME/var/objserv_uni/A_TO_D_GATE_1.store'`

  This property specifies the name of the file that will be used to store data the Gateway has read from the Aggregation layer if it enters store-and-forward mode. (See the previous property for more information on store-and-forward mode.)

- `Gate.Writer.UseBulkInsCmd : TRUE`

  This property (when set to TRUE, as in this case) causes the Gateway to optimise the format of SQL insert commands into batches before sending them to the destination Display ObjectServer. This enables the Display ObjectServer to process the insert commands more efficiently.

- `Gate.Resync.Enable : TRUE`

  This property ensures the Gateway resynchronisation functionality is enabled.

- `Gate.Resync.Type : 'NORMAL'`

  This property specifies that the Gateway resynchronisation type is set to NORMAL. This means that when the Gateway resynchronises, it will delete the entire contents of the Display ObjectServer and then re-copy everything from scratch. This ensures that the Display ObjectServer contains an exact duplicate of what the Aggregation ObjectServer holds after a resynchronisation.

- `Gate.Resync.LockType : 'NONE'`

  This property specifies that the Gateway should not attempt to gain a lock on either the source or destination ObjectServers during the resynchronisation process. This setting allows all inter-tier Gateways to resynchronise simultaneously rather than one at a time. It also means that any Native Event List or IBM Netcool/OMNIbus WebGUI clients this Display ObjectServer is servicing do not experience an interruption in service, which would happen if the Display ObjectServer were locked for the duration of the resynchronisation.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Display Gateway table replication file

This section contains notes on the Aggregation to Display Gateway table replication file named A_TO_D_GATE.tblrep.def. With respect to the status, journal and details tables, the table replication file contains the following:

```
REPLICATE ALL FROM TABLE 'alerts.status'
   USING MAP 'StatusMap'
   SET UPDTOINS CHECK TO FORCED;

REPLICATE ALL FROM TABLE 'alerts.journal'
   USING MAP 'JournalMap';

REPLICATE ALL FROM TABLE 'alerts.details'
   USING MAP 'DetailsMap';
```

This is default, vanilla settings for the replication of these three tables, with one additional option for the replication of the alerts.status table. A note on this option follows below.

NOTES:

- SET UPDTOINS CHECK TO FORCED

  Under standard operation, the Gateway maintains a cache of the events on both the source and target ObjectServers. On processing ObjectServer IDUC, the Gateway differentiates an insert from an update by whether the event to be transferred from the source ObjectServer exists on the destination ObjectServer. If it does not exist on the destination ObjectServer, it sends the event as an INSERT. If it does exist on the destination ObjectServer, it sends the event as an UPDATE.

  The SET UPDTOINS CHECK TO FORCED directive tells the Gateway not to bother checking its cache when processing updates, and simply to send all updates as inserts. Hence the Gateway will not differentiate between an insert and an update, and simply send the events for both categories as inserts.

  By not having to check its cache, this speeds up the transfer of alerts.status IDUC data. Also, it simplifies the handling of the data at destination ObjectServer. An incoming insert at the Display layer will be treated as such if the event does not already exist and get processed by the insert trigger: dsd_new_row. If the event already exists, the insert will be handled by the reinsert trigger: dsd_deduplication — which has specifically been designed for this functionality. There is no need to handle incoming updates from the Aggregation layer.

In contrast to the default table replication file however, all other table replications are uncommented for use including: the user related system tables, the desktop related system tables and the master.servergroups table (used for automatic load-balancing of Native Event List clients).

Note: If any custom tables need to be replicated between the Aggregation and Display layers, replication definitions should be created and appended to the end of this file. Note that a corresponding mapping must also exist in the Gateway's mapping file. See the section entitled *Gateway configuration files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway replication file entries and mappings.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

The Display Gateway mapping file

The Display Gateway mapping file (named `A_TO_D_GATE.map`) provides the following section to insert custom field mappings for the `alerts.status` table:

```
#####################################################################
#
#  CUSTOM alerts.status FIELD MAPPINGS GO HERE
#
#####################################################################



#####################################################################
```

Note: Any custom table mappings should be appended to the end of the mapping file. See the section entitled *Gateway mapping files* in Chapter 6 of this document for more guidelines on the creation of custom Gateway mappings.

The TimeToDisplay field

The *TimeToDisplay* functionality is provided in the standard multitier architecture configuration to provide statistical information as to how long it is taking for events to reach the Display ObjectServers from the time they are initially inserted into either the Collection or Aggregation layers.

This information provided in each event and in the synthetic event (that shows an average `TimeToDisplay` for all current events) is useful for Netcool Administrators and operators alike as an indication of potential IBM Netcool/OMNIbus system overloading or bottlenecks. For example, if lengthy `TimeToDisplay` values are present for events from a single Collection layer ObjectServer only, it would help identify that there is likely an issue with that Collection ObjectServer or its interconnecting Gateway.

The fields `CollectionFirst`, `AggregationFirst` and `DisplayFirst` record the timestamps of when an event first reaches each tier via insert database triggers at each layer. The insert database trigger at the Display layer that sets the `DisplayFirst` field value also performs the calculation for the `TimeToDisplay` field for the current event.

Note: If an event is inserted at the Aggregation layer, the contents of the `CollectionFirst` field is left set to zero. The trigger that calculates the `TimeToDisplay` value however takes a zero value in this field to indicate that it was inserted at the Aggregation layer, and factors this into its calculations.

Every 61 seconds, a temporal trigger called `calculate_time_to_display` iterates over every event with a *"valid"* `TimeToDisplay` value and calculates the average. It then creates/updates a synthetic event in the local Display ObjectServer that notifies the users of the current average `TimeToDisplay` value.

WHAT IS MEANT BY A "VALID" TIMETODISPLAY VALUE?

When the Aggregation to Display Gateway resynchronises, it performs a FULL resynchronisation from the Aggregation ObjectServer. This works by deleting the contents of the Display ObjectServer

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

and recopying all the events afresh. Since the `DisplayFirst` value is set on initial insert into the Display ObjectServer, the Gateway resynchronisation has the side-effect of incorrectly skewing both event and overall average `TimeToDisplay` values upwards. This is because the `CollectionFirst` and `AggregationFirst` fields will still contain their original values but the `DisplayFirst` field will be reset to the current timestamp when the resynchronisation occurs. This will result in an incorrect `DisplayFirst` value, often significantly later than the `CollectionFirst` and `AggregationFirst` fields.

Because of this, a signal trigger called `tag_old_events` exists in the Display ObjectServers to set the `TimeToDisplay` value to 9999999 for all events in the Display ObjectServer immediately after a Gateway resynchronisation. Any events that have this value will appear in the Event List with "N/A" in the `TimeToDisplay` field courtesy of a conversion, and these values will be excluded from any average `TimeToDisplay` calculations.

Any events subsequently inserted into the Display ObjectServer after the Gateway resynchronisation has completed will generate "valid" `TimeToDisplay` values, and will be included in the average `TimeToDisplay` calculations.

Note: For further reading on the *TimeToDisplay* functionality, see the previous section entitled *Display layer ObjectServer triggers* (for specific trigger details) in this document and also the section entitled *The performance triggers* in the *IBM Netcool/OMNIbus 8.1 Installation & Deployment Guide*.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 8 Proxy servers

The function of a Proxy server is to multiplex Probe connections into an ObjectServer. This is achieved by configuring multiple Probes to connect to the Proxy server and then configuring the Proxy server to connect to an ObjectServer.

Connecting Probes have no visibility of the Proxy server and think they are connecting directly to the ObjectServer. This is because the Proxy server does what its name suggests and simply relays communications transparently between the Probe and the ObjectServer.

The following list provides two scenarios where a Proxy server might be deployed:

- If the number of available connections into an ObjectServer is limited, then a Proxy server can be deployed. The Proxy server only consumes one connection within the ObjectServer, regardless of how many Probes connect to the Proxy server. Each Probe, on the other hand, would normally consume one connection per Probe.
- Sometimes, the Probes can be separated from the ObjectServer by a firewall and rules need to be created to allow the connection of the Probes to the ObjectServer. To simplify firewall rule creation and ongoing maintenance, and also to reduce the number of connections through the firewall, a Proxy server may be deployed on the Probe side of the firewall. The Proxy server will only require one connection through the firewall to the ObjectServer, regardless of the number of connected Probes, plus the firewall rule does not need to be updated every time a new Probe is added.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 9 Firewall Bridge server

In a secure environment in which the ObjectServer and the Probes are separated by a firewall, a Firewall Bridge server can be configured so that the Probes can connect to the ObjectServer from outside the secure network.

Note: For more information regarding how to set up a Firewall Bridge server, including some example scenarios, see *Chapter 3. Configuring a firewall bridge server* in the *IBM Netcool/OMNIbus 8.1 Administration Guide*.

Warning: Care must be taken in deploying the Firewall Bridge so that corporate network security policies are not inadvertently breached. In most cases, it is likely that the operation processes of the bridge will have to be reviewed by network security teams, and relevant authorisation obtained before using it.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Chapter 10 The Netcool Process Agent and machine start-up

As mentioned previously in this document, all IBM Netcool/OMNIbus processes should be configured to start and run under Netcool Process Agent control. This is a best practice. Doing so ensures that all processes will be restarted should they fail for any reason, providing an additional degree of resiliency.

Netcool Process Agent configuration file

The default Netcool Process Agent configuration file is in:

`$OMNIHOME/etc/nco_pa.conf` (UNIX)

`%OMNIHOME%\etc\nco_pa.conf` (Windows)

The following list contains some best practice tips in relation to the Netcool Process Agent configuration file:

- Before modifying the default Process Agent configuration file, always make a backup copy of the original file to keep as a reference. A suggested naming convention for the backup copy is: `nco_pa.conf.<date>` — for example: `nco_pa.conf.20110929`
- When adding new processes and/or services to the Process Agent configuration file, ensure the new entries conform to the layout and formatting of the existing default entries. This includes the use of tabs (ie. not *just* space characters) when adding necessary whitespace on individual lines.
- Unless there is a specific reason processes should not auto-start, all processes should be set to start automatically.
- All custom entries in the Netcool Process Agent configuration file must be fully documented, either in-line in the file itself, in the solution detailed design documentation (DDD), or both.

Machine start-up

It is a best practice to configure the Netcool Process Agent to start automatically on machine start up. On UNIX systems, this is configured by running the following setup utility:

```
$OMNIHOME/install/startup/<arch>install
```

For example, on hosts running Linux operating systems, the command is:

```
$OMNIHOME/install/startup/linux2x86install
```

Note: This setup utility should be run as the `root` system user since it needs to create start-up files and soft links under `/etc/`.

On Windows platforms, the Netcool Process Agent should be configured as a Windows Service and set to start automatically on machine start-up. See the *IBM Netcool/OMNIbus 8.1 Installation & Deployment Guide* for guidance on how to do this.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Process Agent security considerations

ObjectServer external procedure actions allow the ObjectServer to execute an external action on the local operating system. The external action request is passed by the ObjectServer to an IBM Netcool Process Agent which executes the external action on the ObjectServer's behalf.

When defining the external action, the Netcool Administrator must specify which user the external action should run as. If the Netcool Process Agent is running as a privileged or super user on the host machine, then an IBM Netcool Administrator could potentially configure external actions to be carried out on the host system as a privileged user — for example: the root user on a UNIX system or as the Local System user on Windows platforms.

This presents a potential security risk to the host system in that an IBM Netcool Administrator user may not be a privileged user on the host system but could potentially carry out privileged operations on the host system anyway.

The way to prevent this scenario is by running the Netcool Process Agent used by the ObjectServer for external actions as a non-privileged user.

Running the Netcool Process Agent as a non-privileged user (UNIX)

To set up an IBM Netcool Process Agent to run as a non-privileged user on UNIX (for example, the local `netcool` user), use the following steps:

- Install the Netcool Process Agent start-up scripts as the `root` user, since it needs to copy the start-up script under the `/etc/` subdirectory and create soft links.
- As the root user, edit the start-up script and modify the line that starts the Netcool Process Agent. For example, on a Linux platform, the start-up script is located in `/etc/init.d/nco`. Locate the line that appears as follows:

```
if [ "$SECURE" = "Y" ]; then
      ${OMNIHOME}/bin/nco_pad -name ${NCO_PA} -authenticate PAM -
secure > /dev/null 2> /dev/null
else
      ${OMNIHOME}/bin/nco_pad -name ${NCO_PA} -authenticate PAM >
/dev/null 2> /dev/null
fi
```

The above commands will run the start-up script as the system user `root`, and hence the Netcool Process Agent will be started as the `root` user. Change the above lines to the following to start the Netcool Process Agent as the `netcool` user instead:

```
if [ "$SECURE" = "Y" ]; then
      su - netcool -c "${OMNIHOME}/bin/nco_pad -name ${NCO_PA} -
authenticate PAM -secure > /dev/null 2> /dev/null"
else
      su - netcool -c "${OMNIHOME}/bin/nco_pad -name ${NCO_PA} -
authenticate PAM > /dev/null 2> /dev/null"
fi
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

This will result in the Netcool Process Agent being started as the local user `netcool`. As a result, the ObjectServer external actions permissions will be limited to what the user `netcool` can do on the host system.

Running the Netcool Process Agent as a non-privileged user (Windows)

To set up an IBM Netcool Process Agent to run as a non-privileged user on Windows (for example, the local `netcool` user), use the following steps:

- Log in to the host machine as an Administrator and install the Netcool Process Agent as a service as normal.
- Open the Services window and double-click the new Netcool Process Agent Service to edit it.
- Click on the tab labelled *"Log On"* and change the *"Log on as:"* radio button selection from *"Local System account"* to *"This account:"*, and then select the local `netcool` user to run the Netcool Process Agent as. Click on OK to finish. You may need to restart the Service if the Service was already auto-started as a privileged user.

This will result in the Netcool Process Agent being started as the local user `netcool`. As a result, the ObjectServer external actions permissions will be limited to what the user `netcool` can do on the host system.

Running the Netcool Process Agent as a privileged user

There are instances however where the Process Agent needs to be run as a privileged user — for example when running the SNMP Probe which needs to open port 162, or when local file authentication is in use and the Process Agent needs to be able to read the `/etc/shadow` file to authenticate users (UNIX only).

In this case, the *primary* Process Agent should be installed and run as the privileged user, and a *secondary* Netcool Process Agent running as a non-privileged user configured to run for the execution of external ObjectServer actions.

The primary Netcool Process Agent should be configured to run on machine start-up in the normal way — via the start-up script (on UNIX) or via the Service (on Windows). The secondary Netcool Process Agent should be configured to run in non-daemon mode as a child process of the primary Netcool Process Agent.

To do this, create an additional process entry in your primary Netcool Process Agent configuration file like the following example:

```
nco_process 'NON_ROOT_PA'
{
   Command '$OMNIHOME/bin/nco_pad -name NON_ROOT_PA -nodaemon -
configfile $OMNIHOME/etc/NON_ROOT_PA.conf' run as 1000
   Host        =     'hostx'
   Managed     =     True
   RestartMsg  =     '${NAME} running as ${EUID} has been restored on
${HOST}.'
   AlertMsg    =     '${NAME} running as ${EUID} has died on
${HOST}.'
   RetryCount  =     0
   ProcessType =     PaPA_AWARE
}
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

NOTES:

- The secondary Netcool Process Agent is set to start as user 1000 (UID) in this example.
- The secondary Netcool Process Agent is set to start with the `-nodaemon` start-up switch. This prevents it from forking to a child process and subsequently allows it to be managed by the primary Netcool Process Agent.
- The secondary Netcool Process Agent will require a configuration file to be set up for it. The file itself needs not contain any processes or services but the file must have the relevant file permissions to enable it to be accessed by the user the secondary Netcool Process Agent is running as.
- The ObjectServer running as a process under the primary Netcool Process Agent needs to have its properties file modified (eg. `AGG_P.props`) to include the authentication details of the secondary Netcool Process Agent including the setting of the properties: `PA.Name`, `PA.Username` and `PA.Password`.

Note: See the section entitled *PA.Name, PA.Username, PA.Password* in *Chapter 4 ObjectServers* in this document for more information regarding the setting of these properties.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Appendix A. Initial requirements gathering checklist

This checklist should be completed during the planning phase of an IBM Netcool/OMNIbus deployment to capture the primary key metrics required to size and scope the architecture.

<table>
<tr><td>Estimated number of managed entities events will come from:</td><td></td></tr>
<tr><td>Estimated total number of events coming from managed entities:</td><td></td></tr>
<tr><td>Estimated total event rate from all managed entities in events per second:</td><td></td></tr>
<tr><td>Estimated event rate categorised by Probe type in events per second:
Probe type: ……………………………………………………
Probe type: ……………………………………………………
Probe type: ……………………………………………………
Probe type: ……………………………………………………
Probe type: ……………………………………………………
Probe type: ……………………………………………………</td><td></td></tr>
<tr><td>Estimated event rate at the ObjectServer (ie. after event discards at Probe level):</td><td></td></tr>
<tr><td>Estimated rate of deduplication:</td><td></td></tr>
<tr><td>Estimated maximum number of standing events in the ObjectServer (ie. after deduplication) and including maximum potential event bursts:</td><td></td></tr>
<tr><td>Estimated maximum number of concurrent users of the system:</td><td></td></tr>
<tr><td colspan="2">High level outline of the event expiry policy that will be implemented:
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………</td></tr>
<tr><td colspan="2">List of components that will connect or interact with the ObjectServer(s) — page …… of …… :
……………………………………………………………………</td></tr>
</table>

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

List of all custom functionality required with the deployment (in pseudocode) — page …… of …… :

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Appendix B. nco_test_gateway.sh

This script is provided as a mechanism for managing a cold stand-by Gateway on a UNIX host. See the section entitled *Cold standby Gateways (UNIX only)* on page 123 for more information regarding its usage.

```
#!/bin/sh
#
# (C) 2004 Micromuse, Inc.  All Rights Reserved.
#
# All Rights Reserved
#
# RESTRICTED RIGHTS:
#
# This file may have been supplied under a license.
# It may be used, disclosed, and/or copied only as permitted
# under such license agreement.  Any copy must contain the
# above copyright notice and this restricted rights notice.
# Use, copying, and/or disclosure of the file is strictly
# prohibited unless otherwise provided in the license
# agreement.
#
# Identifier: nco_test_primary V1.1 05 October 2005
# Updated to allow for resynchronization before giving up
# on sending signals
# Dan Roscigno (Micromuse Inc)
# Original work by Don Wildman (Micromuse Inc).
#
# 25-SEP-2008
# Script modified by Zane Bray (IBM UK Ltd.) to monitor Gateways
# instead.  It works by bringing up a cold-standby backup Gateway
# when it detects that the primary is down and shuts down the backup
# when the primary comes back up again.

# Script to test for Gateway availability.
#

# ObjectServer Gateway names
PRIMARY_GATEWAY='GATE_P'
SECONDARY_GATEWAY='GATE_B'
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
PA_NAME='NCO_PA'
BACKUP_GATEWAY_PROCESS_NAME='BackupGateway'
NCO_USER='ncouser'
NCO_PASS='netcool'

# Set this for number of retries before entering failure mode
RETRIES="5"

# Set this for number of failback retries
FAILBACKRETRIES="5"

# Set this for poll period
POLL="5"

# Set this for failback poll period
FAILBACKPOLL="60"

# Location of nco_ping
NCO_PING=$OMNIHOME/bin/nco_ping

# Environment
if [ -z "$OMNIHOME" ]; then
   logger -p daemon.notice -t FAILOVERMONITOR "OMNIHOME environment
variable not set, using /opt/netcool/omnibus."
   OMNIHOME=/opt/netcool/omnibus
   export OMNIHOME
fi
############################ subroutines
normal_poll() {
   COUNT="0"
   while [ "$COUNT" -lt "$RETRIES" ]
   do
         # Is the Primary responding?
         $NCO_PING $PRIMARY_GATEWAY >/dev/null 2>&1
         if [ "$?" -ne 0 ]; then
               COUNT=`expr $COUNT + 1`
               logger -p daemon.notice -t FAILOVERMONITOR
"$PRIMARY_GATEWAY failed to respond.  Count is $COUNT."
         else
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
                COUNT="0"
          fi
          sleep $POLL
    done
    logger -p daemon.notice -t FAILOVERMONITOR "$PRIMARY_GATEWAY
failed to repond $RETRIES times."
}
############################ End normal_poll
failed_poll() {
    UNAVAILABLE="1"
    while [ "$UNAVAILABLE" -eq "1" ]
    do
          # Is the Primary responding?
          $NCO_PING $PRIMARY_GATEWAY >/dev/null 2>&1
          if [ "$?" -eq 0 ]; then
                UNAVAILABLE="0"
          else
                sleep $POLL
          fi
    done
}
############################ End failed_poll
start_backup() {
    $OMNIHOME/bin/nco_pa_start -server $PA_NAME -user $NCO_USER -
password $NCO_PASS -process $BACKUP_GATEWAY_PROCESS_NAME
}
############################ End send_down
confirm_failover() {
    COUNT="0"
    FAILEDOVER="0"
    while [ "$FAILEDOVER" -eq "0" ]
    do
          if [ "$COUNT" -lt "$FAILBACKRETRIES" ]; then
                logger -p daemon.notice -t FAILOVERMONITOR "Starting
backup Gateway..."
                start_backup
                if [ "$?" -eq 0 ]; then
                      FAILEDOVER="1"
                      logger -p daemon.notice -t FAILOVERMONITOR
"Backup Gateway started."
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
		else
			FAILEDOVER="0"
			COUNT=`expr $COUNT + 1`
			logger -p daemon.notice -t FAILOVERMONITOR
"$SECONDARY_GATEWAY failed to start.  Count is $COUNT."
			sleep $FAILBACKPOLL
		fi
	fi
  done
  if [ "$FAILEDOVER" -eq "0" ]; then
	logger -p daemon.notice -t FAILOVERMONITOR "Neither
$PRIMARY_GATEWAY nor $SECONDARY_GATEWAY are running!  Please correct
manually."
  fi
}
############################ End confirm_failover
shutdown_backup() {
  $OMNIHOME/bin/nco_pa_stop -server $PA_NAME -user $NCO_USER -
password $NCO_PASS -process $BACKUP_GATEWAY_PROCESS_NAME
}
############################ End send_up
confirm_failback() {
  COUNT="0"
  FAILEDBACK="0"
  while [ "$FAILEDBACK" -eq "0" ]
  do
	if [ "$COUNT" -lt "$FAILBACKRETRIES" ]; then
		logger -p daemon.notice -t FAILOVERMONITOR "Shutting
down backup Gateway..."
		shutdown_backup
		if [ "$?" -eq 0 ]; then
			FAILEDBACK="1"
			logger -p daemon.notice -t FAILOVERMONITOR
"Backup Gateway shut down."
		else
			FAILEDBACK="0"
			COUNT=`expr $COUNT + 1`
			logger -p daemon.notice -t FAILOVERMONITOR
"$SECONDARY_GATEWAY failed to respond.  Count is $COUNT."
			sleep $FAILBACKPOLL
		fi
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

```
        fi
   done
   if [ "$FAILEDBACK" -eq "0" ]; then
        logger -p daemon.notice -t FAILOVERMONITOR "Both
$PRIMARY_GATEWAY and $SECONDARY_GATEWAY are running!  Please correct
manually."
   fi
}
############################ End confirm_failback
# Script starts here
while [ "1" -eq "1" ]
do
   logger -p daemon.notice -t FAILOVERMONITOR "Entering normal
polling cycle."
   normal_poll

   logger -p daemon.notice -t FAILOVERMONITOR "Bringing up backup
Gateway."
   confirm_failover

   logger -p daemon.notice -t FAILOVERMONITOR "Entering failure-mode
polling cycle."
   failed_poll

   logger -p daemon.notice -t FAILOVERMONITOR "Shutting down backup
Gateway."
   confirm_failback
done
```


© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Notices

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 (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation

Licensing

2-31 Roppongi 3-chome, Minato-ku

Tokyo 106-0032, 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

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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. 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

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

These terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- *AIX*
- *developerWorks*
- *IBM*
- *Lotus*
- *Netcool*
- *Tivoli*
- *WebSphere*

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.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, *Windows*, *Windows NT*, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Intel is a registered trademark of Intel Corporation, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

© Copyright IBM Corporation 2012, 2013, 2014, 2017.

Licensed Materials – Property of IBM