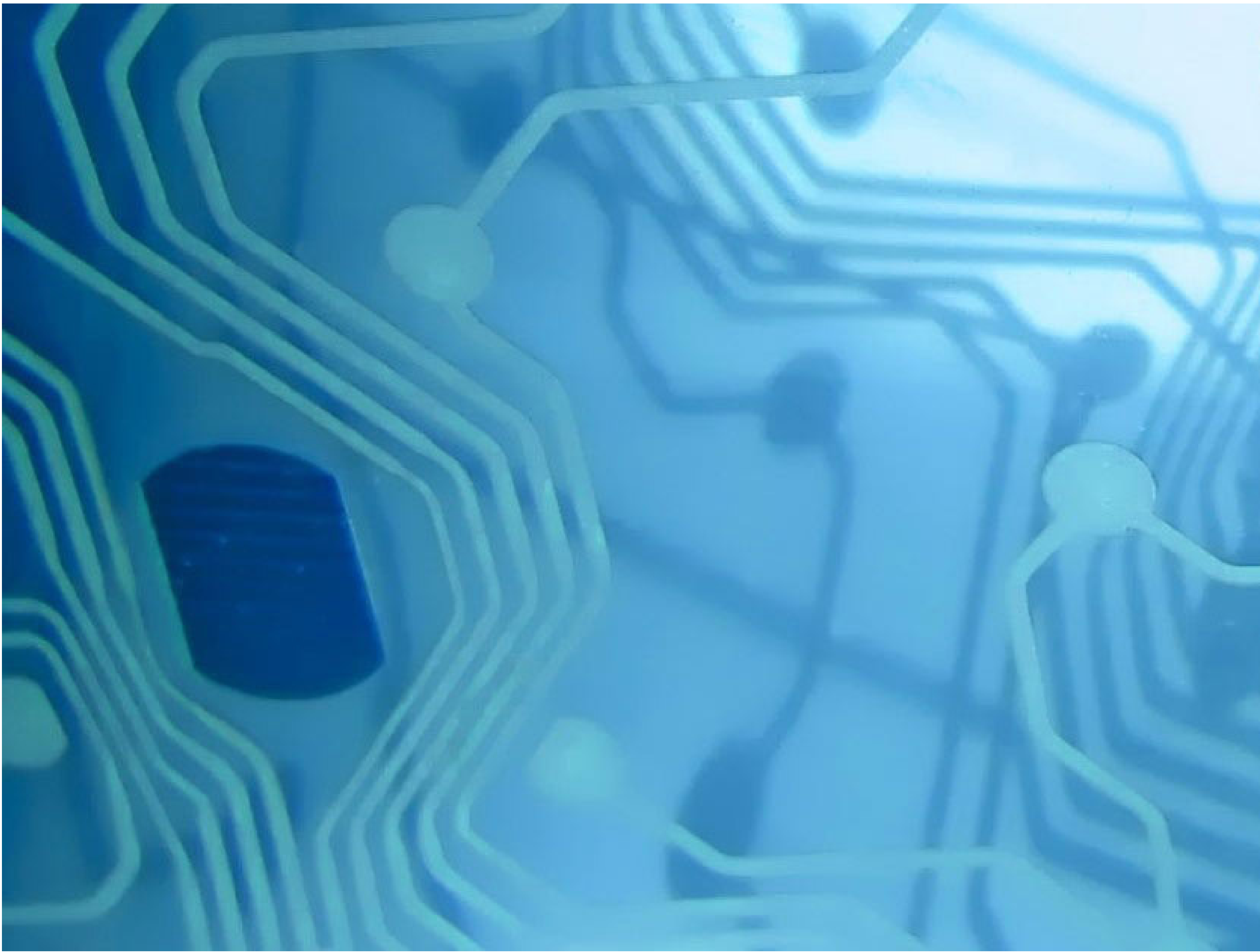
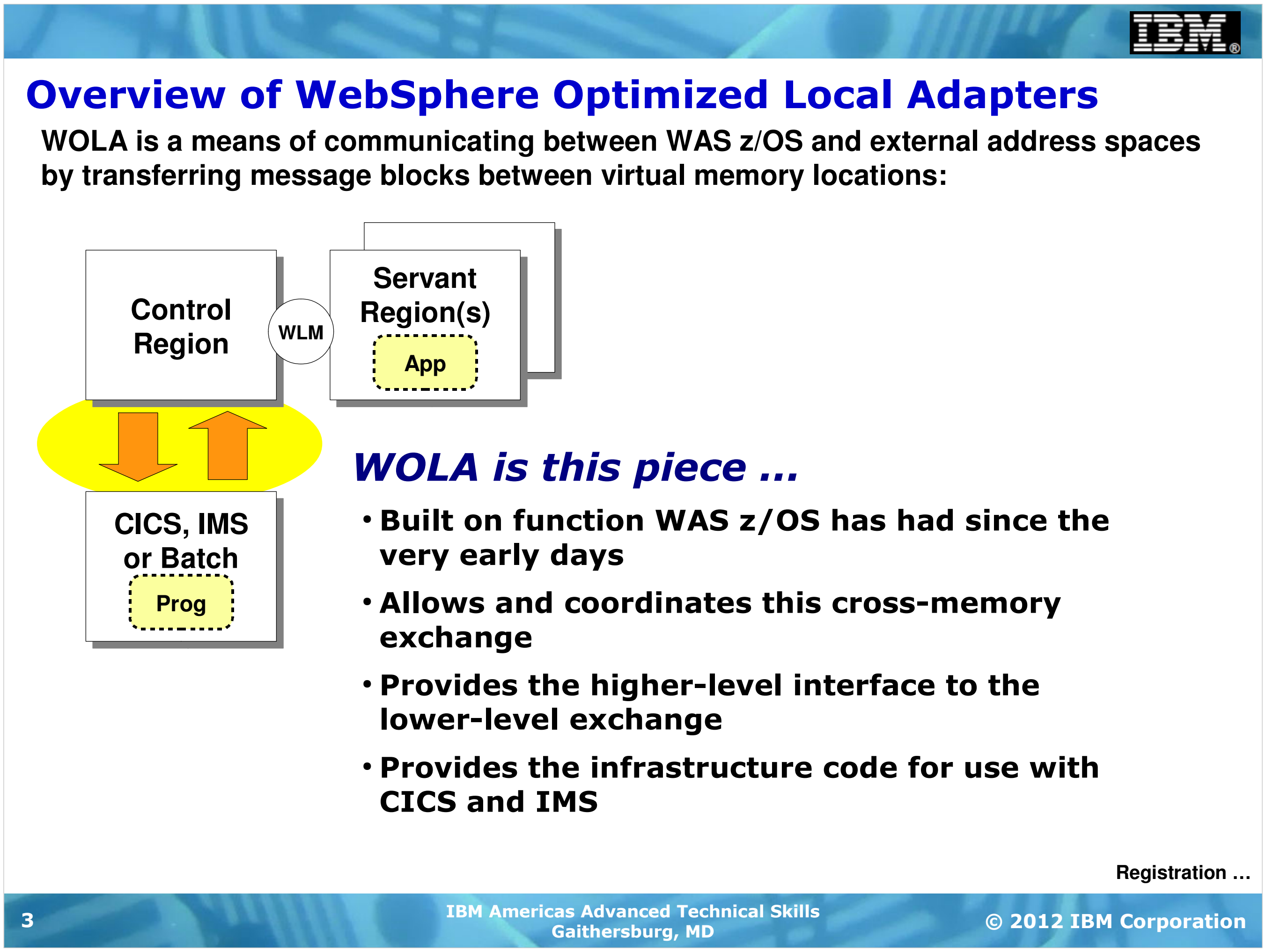
WebSphere Application Server V8.5 for z/OS

WBSR85

Unit 6 - WOLA

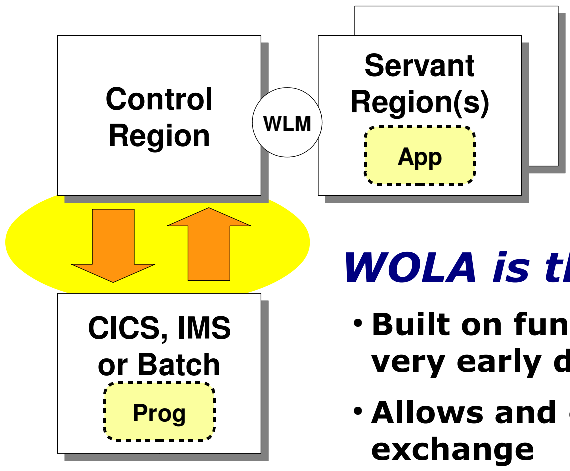


This page intentionally left blank



Overview of WebSphere Optimized Local Adapters

WOLA is a means of communicating between WAS z/OS and external address spaces by transferring message blocks between virtual memory locations:



WOLA is this piece …

- **Built on function WAS z/OS has had since the very early days**
- **Allows and coordinates this cross-memory exchange**
- **Provides the higher-level interface to the lower-level exchange**
- **Provides the infrastructure code for use with CICS and IMS**

Registration …

WebSphere Optimized Local Adapters (WOLA) is a means of communicating between a WAS z/OS application server and another address space using cross-memory transfer of message blocks.

The mechanism used has been in WAS z/OS since the beginning of the product. The designers of WAS z/OS took advantage of a z/OS service to move messages between address spaces using cross-memory copy. WAS z/OS uses this still -- any IIOP messages between servers in a cell on an LPAR makes use of this memory copy function. WOLA is an externalization of this function so other products can use it to acess WAS z/OS.

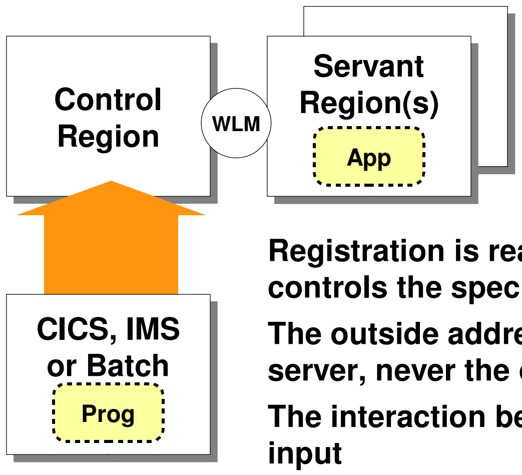
WOLA supports CICS, IMS, batch programs, UNIX System Services processes and Airline Line Control (ALCS) interaction.

The chief benefit of WOLA is that it's efficient and has very low inherent latency. The low latency is important whenenver there is repetition. A one second latency for each of only 10 invocations can be endured; a one second latency suffered a million times adds up.

For WOLA to work there's a bit of infrastructure support that's needed in the WAS z/OS side and the CICS, IMS or batch program side. In this unit we'll look at CICS and batch programs, and in so doing we'll cover the key points of WOLA that will allow you to see how it works and how it can be of value to you.

Registration

An important key concept is "registration" ... the construction of the cross-memory linkage into the WAS z/OS application server:



Serves as the cross-memory "pipe" over which exchanges occur

Registration is really a set of control blocks that permits and controls the specific cross-memory exchanges

The outside address space always registers into the WAS z/OS server, never the other way around

The interaction between CR and SR is the same as for any form of input

Any given WAS z/OS server may have multiple registrations into it

Registration is accomplished in several ways:

- **A supplied CICS control transaction**
- **The BBOA1REG API**

Outbound vs. Inbound ...

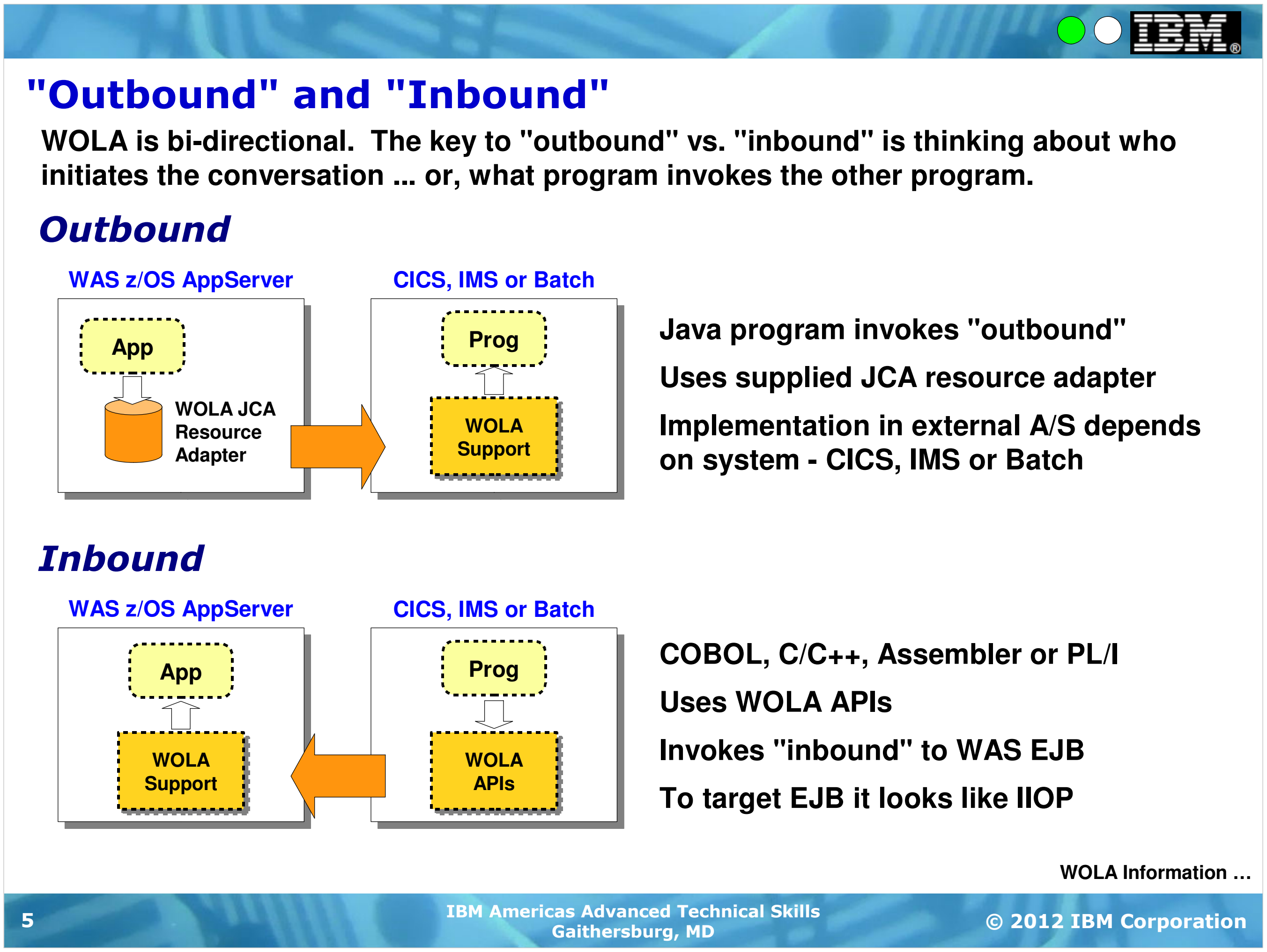
"Registration" is a key starting concept we need to get into the discussion. Before any WOLA communications can take place between WAS z/OS and an address space such as CICS, IMS or Batch, a WOLA "registration" must be established. The "registration" is the cross-memory path over which the exchanges take place.

In reality registration really involves the construction of a set of control blocks that define the settings for the WOLA exchange. When an outside address space registers into a WAS z/OS server, WAS creates these control blocks above the 2GB bar.

Key Point: the outside address space always initiates the registration. It's always the outside address space that "registers into" the WAS z/OS server.

Multiple registrations are permitted, so a given WAS z/OS server may communicate to multiple outside address spaces.

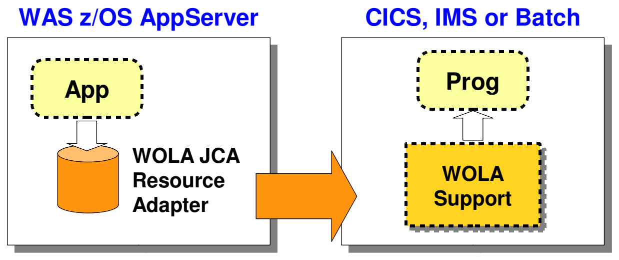
There are two ways this registration takes place -- with a supplied CICS control transaction ("BBOC") or with the `BBOA1REG` native API. We'll discuss both of those in this unit.



"Outbound" and "Inbound"

WOLA is bi-directional. The key to "outbound" vs. "inbound" is thinking about who initiates the conversation ... or, what program invokes the other program.

Outbound

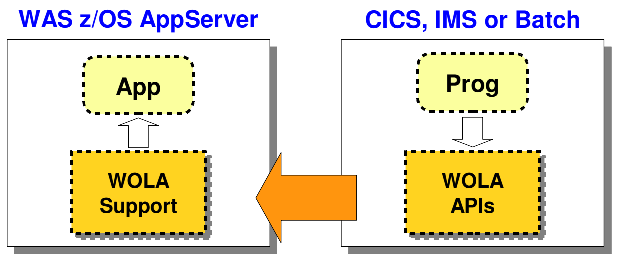


Java program invokes "outbound"

Uses supplied JCA resource adapter

Implementation in external A/S depends on system - CICS, IMS or Batch

Inbound



COBOL, C/C++, Assembler or PL/I

Uses WOLA APIs

Invokes "inbound" to WAS EJB

To target EJB it looks like IIOP

WOLA Information ...

5 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

The best way to start the conversation about WOLA is to establish the concept of the direction of invocation ... "outbound" from WAS to the external address space, or "inbound" from the external address space into WAS.

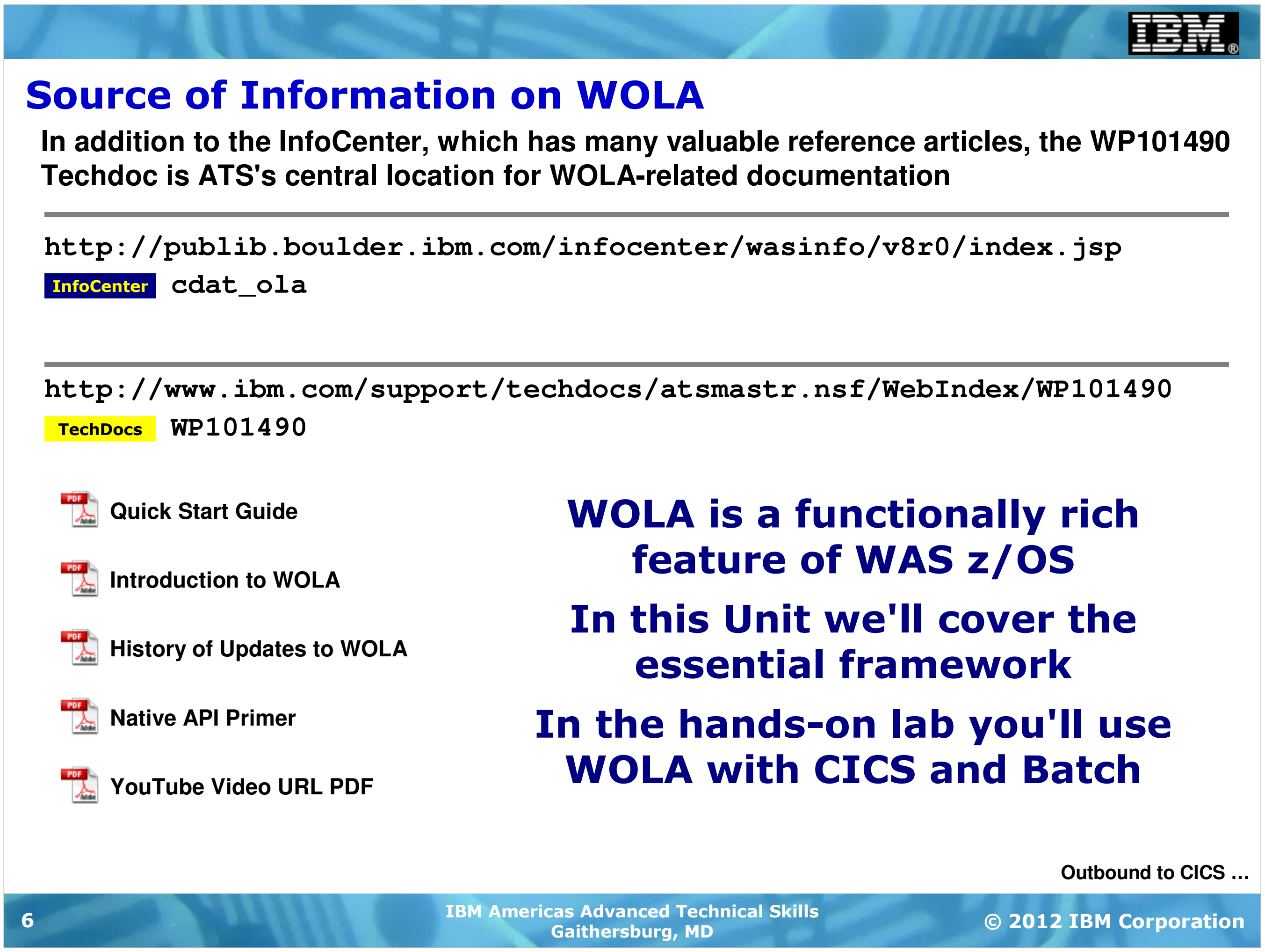
This is really a story of who initiates the conversation ... or who invokes who.

In the outbound model, the Java program in WAS z/OS initiates by invoking outbound to the target program in either CICS, IMS or Batch. In this case the Java program in WAS z/OS makes use of a supplied JCA resource adapter. So to the Java program WOLA looks very much like any JCA resource adapter, such as CTG. The WOLA JCA resource adapter uses the same "common client architecture" (CCI) as CTG, but there getter and setter methods for WOLA are slightly different. So while very similar, it's not identical to CTG for Java programs.

When going outbound the external address space has to have some knowledge of WOLA and how to handle the calls coming over. How this is implemented is different depending on whether it's CICS, IMS or Batch. In this workshop we're going to focus on CICS as the target for outbound calls. In that case the "WOLA Support" box in the picture above becomes a set of CICS functions supplied by WAS that get installed into the CICS region. We'll cover in some detail what those things are.

Note: the response to an outbound call simply flows back across the WOLA connection. The concept of "outbound" and "inbound" focuses on who *starts* the conversation ... the *response* flows back as it would with any conversational connection.

The reverse of "outbound" is "inbound" ... the program outside of WAS seeks to reach into WAS z/OS and invoke a target EJB. In this case the outside program, written in either COBOL, C/C++, High Level Assembler or PL/I, makes use of the supplied WOLA APIs to communicate inbound. The WOLA JCA Resource Adapter is not used for inbound calls ... the inbound calls are handled by built-in WOLA support provided with WAS z/OS. The target EJB has a few WOLA-specific requirements, but in general it's simply a stateless EJB that sees an IIOP call come to it.



Source of Information on WOLA

In addition to the InfoCenter, which has many valuable reference articles, the WP101490 Techdoc is ATS's central location for WOLA-related documentation

`http://publib.boulder.ibm.com/infocenter/wasinfo/v8r0/index.jsp`

InfoCenter `cdat_ola`

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101490`

TechDocs `WP101490`

- **Quick Start Guide**
- **Introduction to WOLA**
- **History of Updates to WOLA**
- **Native API Primer**
- **YouTube Video URL PDF**

WOLA is a functionally rich feature of WAS z/OS

In this Unit we'll cover the essential framework

In the hands-on lab you'll use WOLA with CICS and Batch

Outbound to CICS ...

6 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

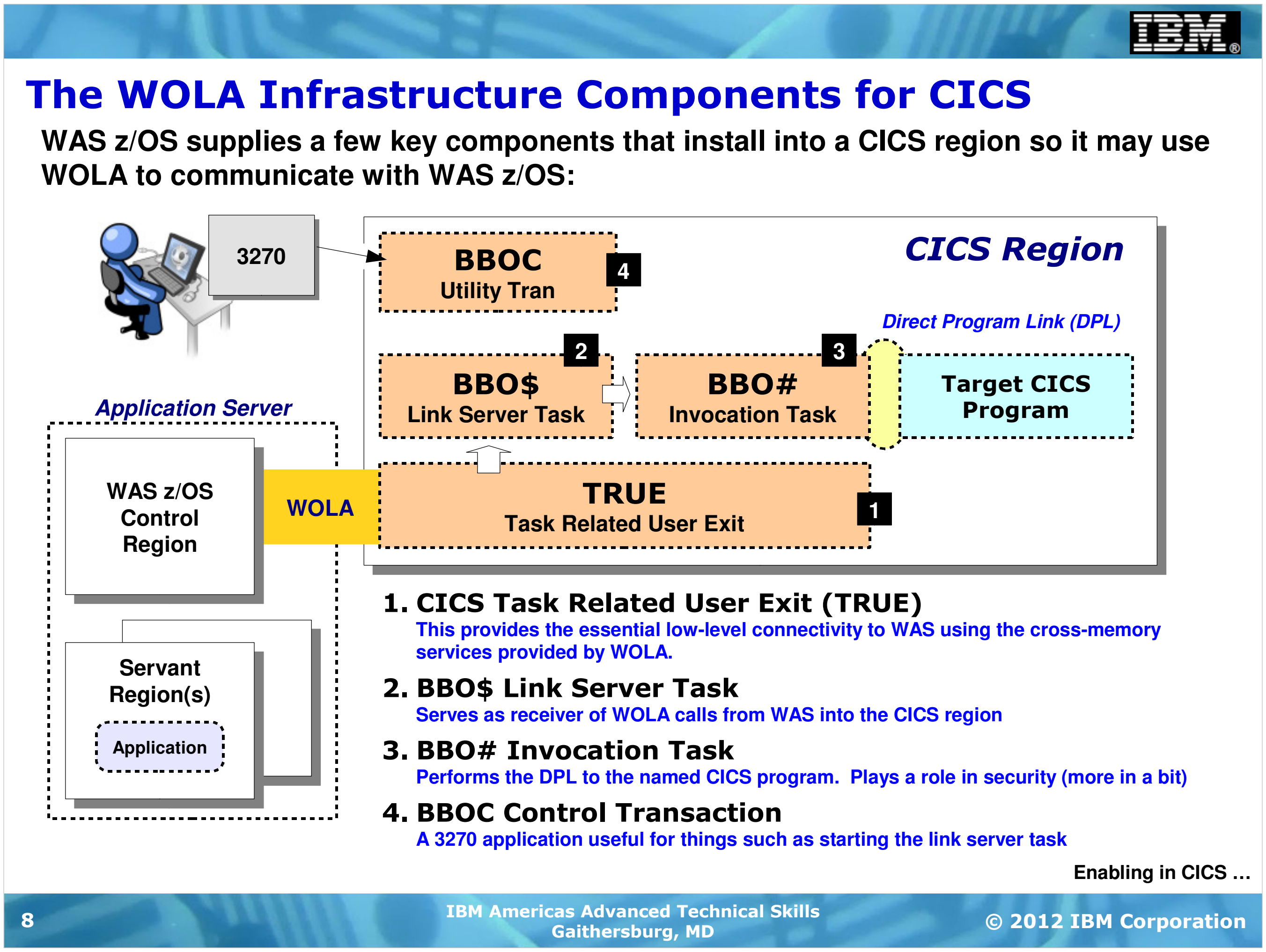
WOLA is a fairly broad topic encompassing a fair deal of concepts as well as lower-level details. In the relatively short time we have for this unit we can't adequately cover it all. But we can give you the key concepts and point you to the sources of detailed information for the function.

The product Information Center (InfoCenter) is a key resource for reference and configuration information. There are many articles on WOLA. The "starting point" is the one we show on the chart above. The easiest way to search and find a specific topic is to search on the keyword string for the article file name.

The Techdoc WP101490 serves as our central repository for all things WOLA. It has many documents contained within it. A few of the documents are shown on the chart above.

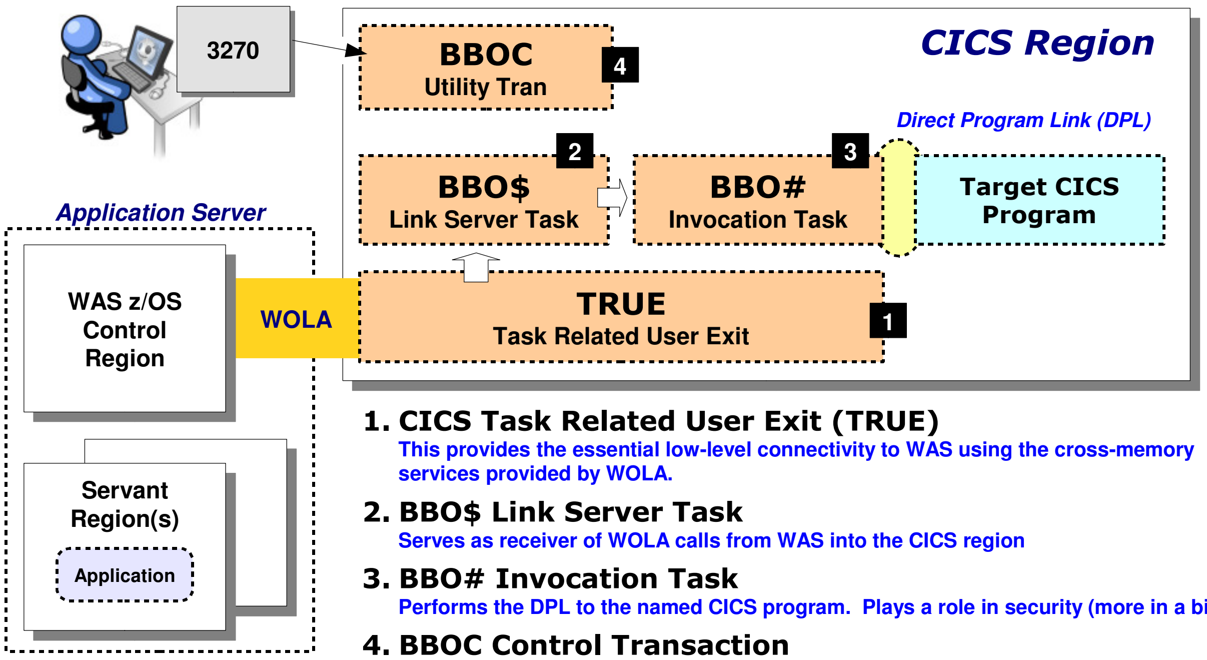
Outbound to CICS

Using the Supplied CICS Link Server Task



The WOLA Infrastructure Components for CICS

WAS z/OS supplies a few key components that install into a CICS region so it may use WOLA to communicate with WAS z/OS:



1. **CICS Task Related User Exit (TRUE)**
   This provides the essential low-level connectivity to WAS using the cross-memory services provided by WOLA.
2. **BBO$ Link Server Task**
   Serves as receiver of WOLA calls from WAS into the CICS region
3. **BBO# Invocation Task**
   Performs the DPL to the named CICS program. Plays a role in security (more in a bit)
4. **BBOC Control Transaction**
   A 3270 application useful for things such as starting the link server task

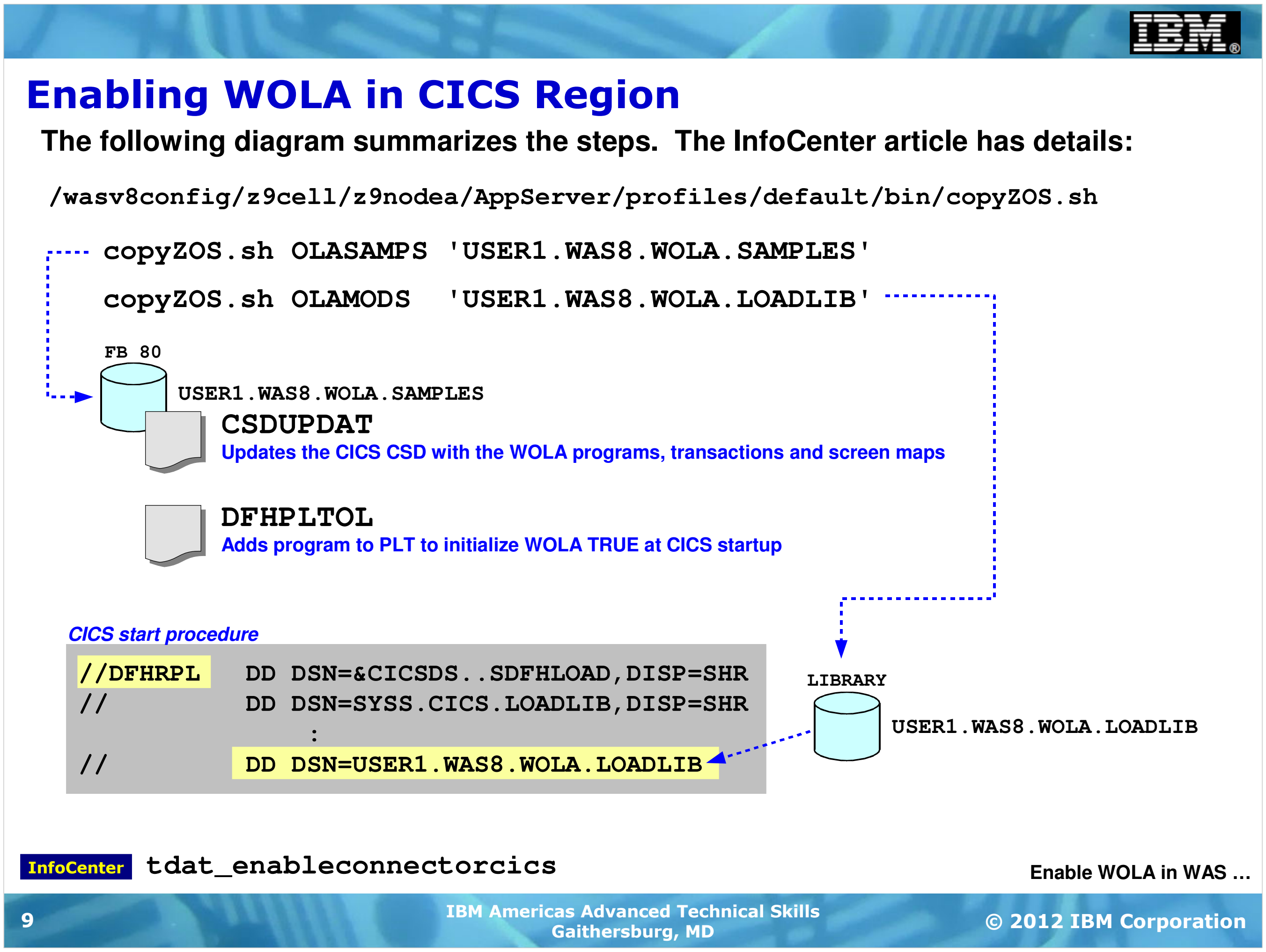
Enabling in CICS …

8 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

For WOLA to work with CICS there needs to be a few pieces of WOLA infrastructure installed and enabled in the CICS region. The chart above illustrates the architecture. The numbered blocks in the picture correspond to the notes here:

1. The lowest-level support in CICS for WOLA is a supplied "Task Related User Task," or TRUE. The TRUE architecture has been a part of CICS for many years. The designers of WOLA wanted to make use of existing CICS architectural support so they implemented this low-level support as a TRUE.
2. For calls flowing outbound from WAS into the CICS region the BBO$ Link Server Task handles the calls and begins the process of invoking the named CICS program.

   Note: it is possible to make use of WOLA and not employ the BBO$ Link Server task. But that implies (a) taking on more of the programming burden yourself, and (b) losing some transaction and security benefits. The next chart highlights those points.
3. The BBO# Link Invocation Task is called by the BBO$ Server task. The invocation task is what performs the DPL to the named CICS program. The reason this function is separate from the server task is because when the user ID on the WAS thread is asserted into CICS there will result a separate instance of the invocation task. Each instance carries the ID asserted into CICS from WAS. The DPL of the named CICS program is made under the identity of the asserted ID. However, if you choose to *not* assert the user ID into CICS then only one invocation task may be used.
4. The BBOC control transaction is a 3270 program that allows you the ability to control elements of the WOLA infrastructure such as starting and stopping the Link Server Task, or starting and stopping the TRUE itself.



Enabling WOLA in CICS Region

The following diagram summarizes the steps. The InfoCenter article has details:

```
/wasv8config/z9cell/z9nodea/AppServer/profiles/default/bin/copyZOS.sh

copyZOS.sh OLASAMPS 'USER1.WAS8.WOLA.SAMPLES'

copyZOS.sh OLAMODS  'USER1.WAS8.WOLA.LOADLIB'
```

FB 80
USER1.WAS8.WOLA.SAMPLES
CSDUPDAT
Updates the CICS CSD with the WOLA programs, transactions and screen maps
DFHPLTOL
Adds program to PLT to initialize WOLA TRUE at CICS startup
LIBRARY
USER1.WAS8.WOLA.LOADLIB

CICS start procedure

```
//DFHRPL    DD DSN=&CICSDS..SDFHLOAD,DISP=SHR
//          DD DSN=SYSS.CICS.LOADLIB,DISP=SHR
                  :
//          DD DSN=USER1.WAS8.WOLA.LOADLIB
```

InfoCenter **tdat_enableconnectorcics**

Enable WOLA in WAS …

The tasks needed to enable WOLA in a CICS region are relatively simple for the CICS system programmer.

WAS z/OS ships in its UNIX file system a set of WOLA sample jobs and a set of WOLA modules. Further, WAS z/OS provides a shell script utility that will copy those file system files out to MVS data sets. That shell script is called `copyZOS.sh` and is found in the `/bin` directory. The shell script assumes the target MVS data sets are pre-allocated (help for allocation parameters is provided if you invoke the shell script with a `-?` switch). Then the shell script may be used to copy the files from the USS file system to the MVS data set.

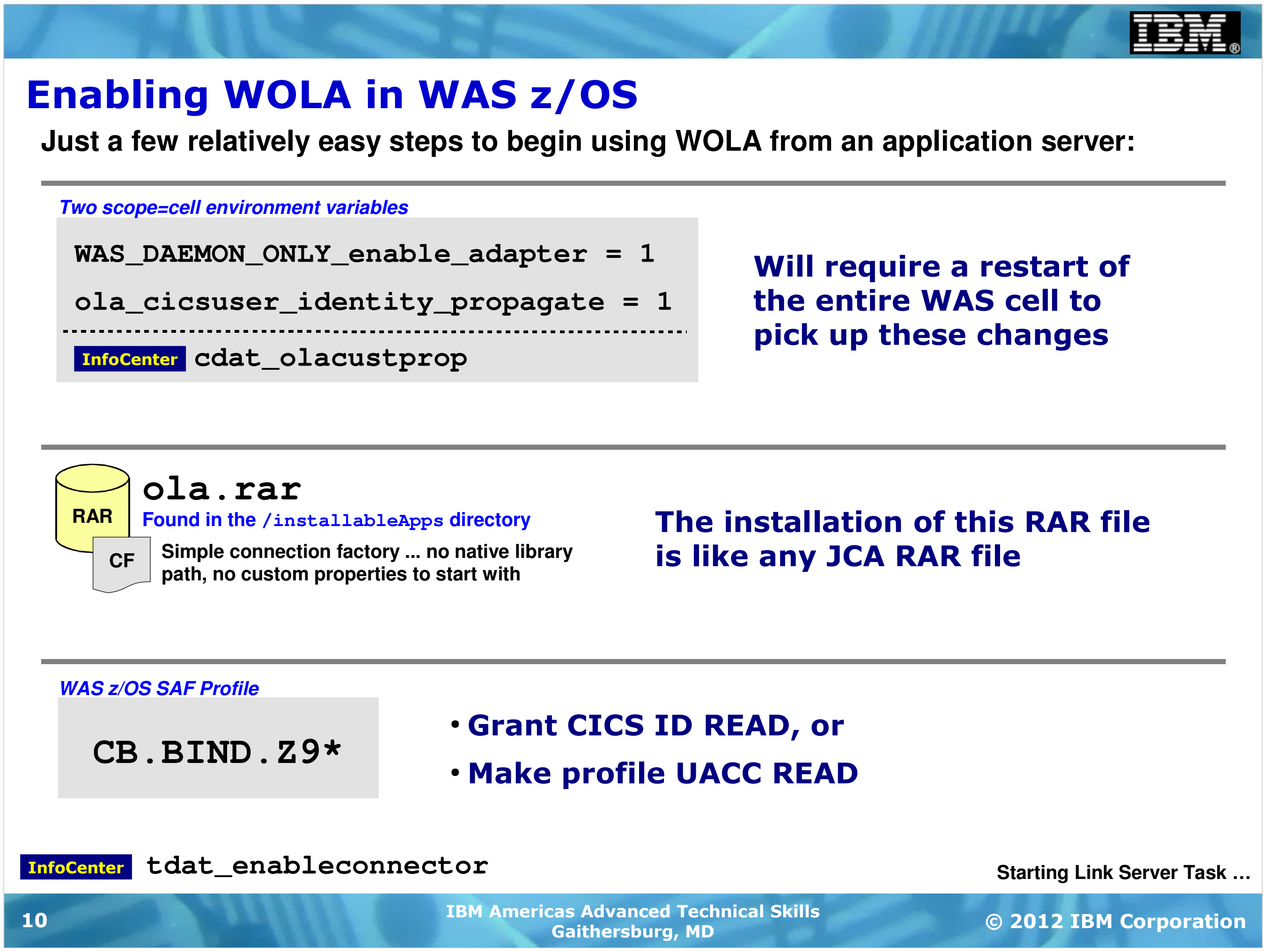
Once copied, there are two sample jobs that will update CICS:

- `CSDUPDAT` - this will update the CICS CSD with information about the WOLA programs, transactions and screen maps. This is required for WOLA to work.

  Note: the JCL that wrappers the commands that update the CSD need to be customized for your particular installation.

- `DFHPLTOL` -- this is optional. It adds a program to the CICS PLT (program load table) so the WOLA TRUE is started automatically when the CICS region is started. You may start the TRUE using a BBOC command if you would rather not have it started automatically.

The other `copyZOS.sh` function copies the WOLA modules out to a `LIBRARY` data set. This is then concatenated on the CICS start procedure `DFHRPL DD` statement to provide CICS access to those modules.



Enabling WOLA in WAS z/OS

Just a few relatively easy steps to begin using WOLA from an application server:

Two scope=cell environment variables

```
WAS_DAEMON_ONLY_enable_adapter = 1
ola_cicsuser_identity_propagate = 1
```

InfoCenter `cdat_olacustprop`

Will require a restart of the entire WAS cell to pick up these changes

RAR CF **`ola.rar`**

Found in the `/installableApps` directory

Simple connection factory ... no native library path, no custom properties to start with

The installation of this RAR file is like any JCA RAR file

WAS z/OS SAF Profile

```
CB.BIND.Z9*
```

- **Grant CICS ID READ, or**
- **Make profile UACC READ**

InfoCenter `tdat_enableconnector`

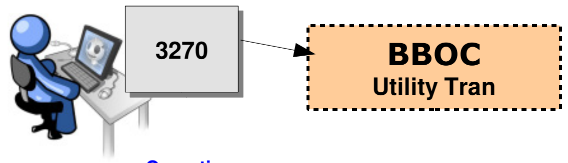
Starting Link Server Task ...

Enabling WOLA in the WAS z/OS runtime environment is relatively simple. Three basic steps are needed:

1. Two cell-level environment variables are involved -- one turns on a switch that enables the WAS z/OS Daemon server to recognize it must begin providing WOLA coordination services, and the other provides the ability to propagate CICS identity into WAS when the flow is "inbound" to CICS. The first is required; the second optional depending on what you are planning to do. Because the first variable involves the Daemon server, it requires the restart of the entire WAS cell (on that LPAR).
2. WOLA comes with a JCA resource adapter much like the CICS Transaction Gateway resource adapter. The WOLA RAR file installs very much like any RAR file installs. The definition of a connection factory is what provides applications the path to access WOLA. We'll see how Java applications in WAS z/OS make use of this JCA resource adapter to access WOLA in an upcoming chart.
3. The final piece is a potential update to a WAS z/OS RACF profile. To understand why this is necessary you first must understand that a key activity in using WOLA is something called "registration." That activity involves the external address space "binding" to the control region of the named application server. It would not be good to allow anything to bind ... some security mechanism is needed. And for WAS z/OS that security mechanism is the CBIND profile ... specifically, the CB.BIND profile. For an external address space to successfully register into a WAS server, the identity of that MVS address space must have READ access to the CBIND profile. There are two ways to providet this -- grant the ID specific READ access, or make the profile itself "Universal Access" (UACC) of READ.

Starting the WOLA Link Server Task in CICS

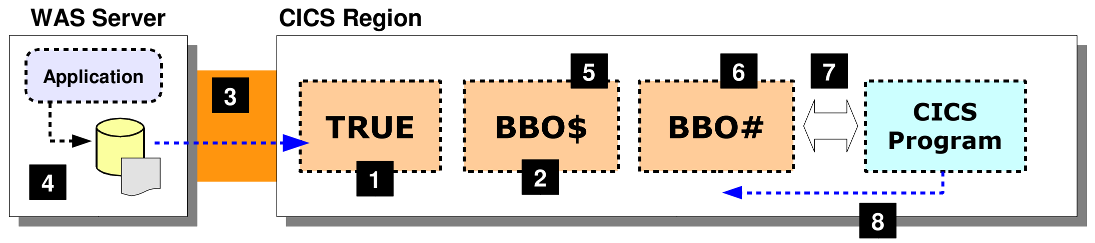
This performs two roles -- it initates the registration into the WAS server, and it prepares the Link Server to accept requests from the application in WAS:



Operation · Register Name · Cell SHORT · Node SHORT · Server SHORT

```
BBOC START_SRVR RGN=CICSXREG DGN=Z9CELL NDN=Z9NODEA SVN=Z9SR01
                        SVC=*   MNC=1   MXC=10   TXN=N   SEC=N   REU=Y
```

Accept any service name · Minimum Connections · Maximum Connections · Propagate Transaction? · Propagate Security? · Reuse BBO#?



See notes for explanation of numbered blocks

InfoCenter **rdat_cics**

Java application considerations ...

With WOLA enabled in the CICS region and the WAS environment, the BBO$ Link Server task may be started in the CICS region to prepare for WOLA calls coming over from WAS.

Note: this assumes the TRUE is started. That may be done automatically with the update to the PLT as performed by the sample job `DFHPLTOL`, or with the `BBOC START_TRUE` command.

The starting of the Link Server task performs the all-important act of "registering" into the named WAS z/OS application server. The registration carries a name, which is important because the application on the WAS side needs to specify *which* registration to talk over. It is possible to have multiple registrations into a given WAS z/OS server, so being able to specify which to use is necessary.

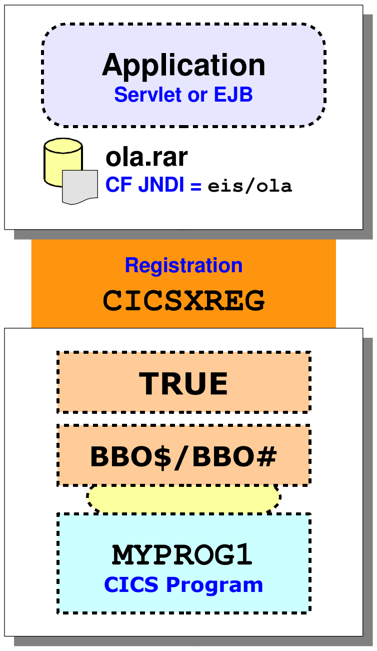
The syntax of the command is shown on the chart, with a pointer to the `rdat_cics` InfoCenter page that provides more detail. Note that you must specify the cell, node and server short names so the registration has the needed detail to register into the named application server.

The numbered blocks in the chart correspond to the following notes:

1. The TRUE is started with the PLT update or manually with `BBOC START_TRUE`.
2. The BBO$ link server task is started
3. The registration into the named WAS application server is made
4. The application uses the installed WOLA resource adapter and defined connection factory to access the WOLA mechanism. The application names the registration to use and the service to invoke. The service is the CICS program to invoke.
5. The request flows into the BBO$ link server task
6. BBO$ creates an instance of the BBO# invocation task
7. The BBO# invocation task performs a direct program link of the named service
8. The response flows back to the WAS application

Java Application Considerations

For outbound use of WOLA to CICS using the Link Server Task the following considerations come into play:



```
Context ctx = new InitialContext();
ConnectionFactory cf
            1 = ctx.lookup("java:comp/env/eis/ola");
ConnectionSpecImpl csi = new ConnectionSpecImpl();
csi.setRegisterName ("CICSXREG"); 2
Connection con = cf.getConnection(csi); 3


Interaction int = con.createInteraction();
InteractionSpecImpl isi = new InteractionSpecImpl();
isi.setServiceName("MYPROG1"); 4
int.execute(isi, data);
```

Either COMMAREA or CICS channel and container. If channel and container, see InfoCenter `rdat_cics`

InfoCenter `tdat_connect2wasapp, tdat_useoutboundconnection`

RA failover ...

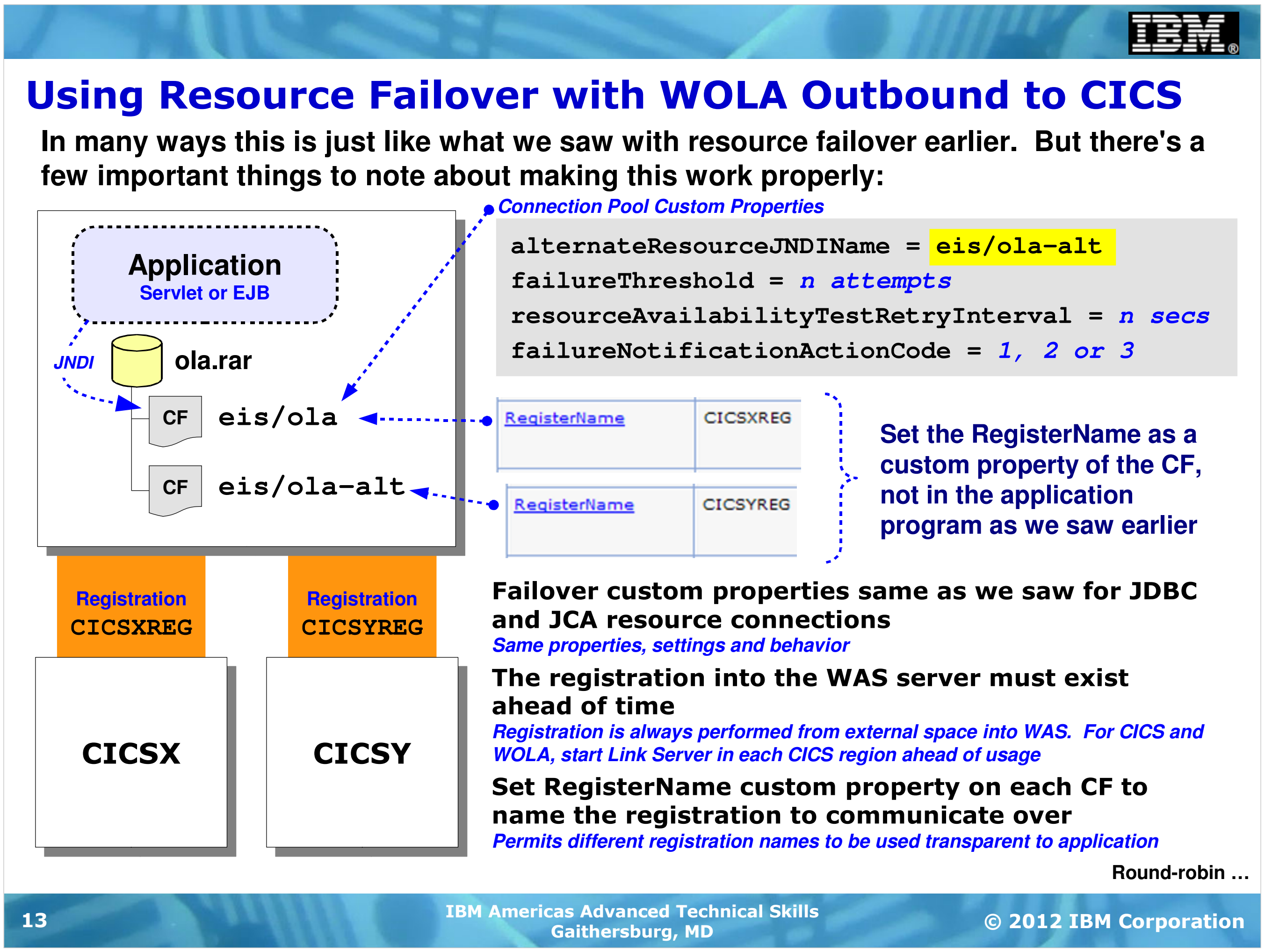
On the WAS z/OS side of this will be a Java application that wishes to call to CICS over the WOLA pipe. That Java application may be in the form of a servlet or EJB. The application is largely shielded from WOLA by the JCA resource adapter that's used to access the lower-level WOLA function. But there are WOLA specific values that must be supplied to the Common Client Interface (CCI) methods implemented in the JCA.

Imagine a setup like what's shown in the chart -- WAS z/OS using WOLA to communicate with a CICS region. The enablement activities are set in both WAS and CICS. The TRUE is started in CICS and the registration into WAS has been successfully performed with the `BBOC START_SRVR` command. The CICS program that will be invoked is `MYPROG1`.

There are two key Java activities -- one is using the `ConnectionSpecImpl()` method to name the registration to communicate over, and the second is using the `InteractionSpecImpl()` method to name the CICS program to invoke.

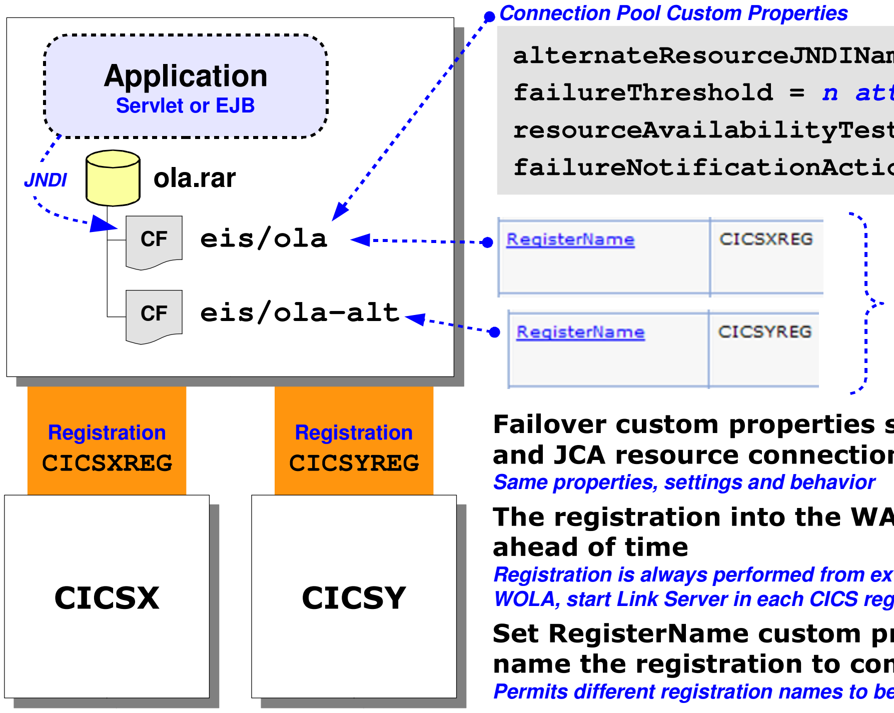
1. The application first does a JNDI lookup of the connection factory. The sample above shows that value as hard-coded but it can be symbolically referenced and resolved at time of deployment.
2. A connection spec implementation object is created and the registration name is set. This value must match that of a valid registration made into the WAS server by the CICS region.
3. A connection object is created
4. An interaction spec implementation object is created, the service name (which is the target CICS program to invoke) is set and the interaction spec is invoked with the data passed.

WOLA itself does not care about the layout of the data ... to WOLA it's just a chunk of memory to be copied. But of course CICS and the CICS program cares, so the *layout* of that data is important. When using the link server task that data may be in the form of a COMMAREA or a container within a channel. There exists tooling support in IBM Rational Application Developer (RAD) to import a COPYBOOK so the data layout created adheres to the intended target.



Using Resource Failover with WOLA Outbound to CICS

In many ways this is just like what we saw with resource failover earlier. But there's a few important things to note about making this work properly:



Connection Pool Custom Properties

```
alternateResourceJNDIName = eis/ola-alt
failureThreshold = n attempts
resourceAvailabilityTestRetryInterval = n secs
failureNotificationActionCode = 1, 2 or 3
```

RegisterName	CICSXREG

RegisterName	CICSYREG

Set the RegisterName as a custom property of the CF, not in the application program as we saw earlier

Failover custom properties same as we saw for JDBC and JCA resource connections
Same properties, settings and behavior

The registration into the WAS server must exist ahead of time
Registration is always performed from external space into WAS. For CICS and WOLA, start Link Server in each CICS region ahead of usage

Set RegisterName custom property on each CF to name the registration to communicate over
Permits different registration names to be used transparent to application

Round-robin …

13 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

Back in the JNDI and CICS sections we saw a new feature for WAS V8 having to do with resource failover and failback. Given that WOLA uses a JCA resource adapter for outbound calls, does that mean it too may participate in this resource failover function? The answer is "yes."

Just as with JDBC and CICS, multiple connection factories need to be defined -- one to serve as the primary and another to serve as the alternate. The same connection pool custom properties apply as did for JDBC and JCA, and the same behavior applies to each custom property.

But there are three things we wish to highlight:

- The WOLA registrations must be established and in place for both the primary and the alternate. Registration is *always* initiated from the external address space into the WAS z/OS server. WAS itself can't initiate a registration to a CICS region. So for this failover to work the alternate registration pipe must be in place and ready to go.
- The registration name must be specified as a custom property on the connection factory, not within the application as we showed on the earlier chart. When the registration name is specified as a custom property on the CF the application no longer has to use the ConnectionSpecImpl object to name the registration. Then when a failover from the primary connection pool to the alternate connection pool is needed, that failover may be transparent to the application. The failover occurs and the registration name on the custom property is used to flow the failover requests across the alternate registration.

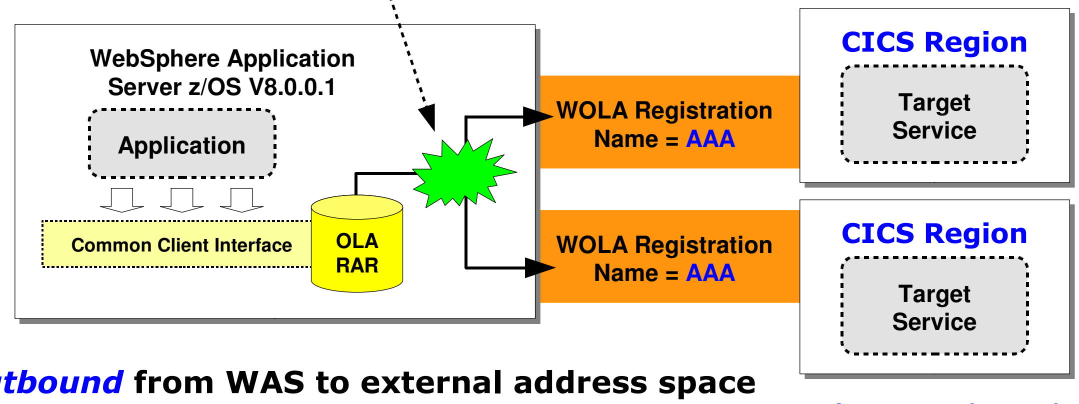
Why would you use this? For CICS in particular the most obvious use-case is when the CICS region you're connecting to is a CICS Gateway Region. This protects against the loss of the primary gateway region while still permitting gateway access to application owning regions in elsewhere on the LPAR or in the Sysplex.

V8.0.0.1 and WOLA Round-Robin

The 8.0.0.1 fixpack brought new WOLA function, including ability to round-robin between multiple CICS regions registered into the server with the same name:

Environment Variable

`ola_locate_service_search_algorithm`	1	The last external address space to register in gets work
	2	Round-robin across like-named registrations


WebSphere Application Server z/OS V8.0.0.1
Application
Common Client Interface
OLA RAR
WOLA Registration Name = AAA
WOLA Registration Name = AAA
CICS Region
Target Service
CICS Region
Target Service

Any supported external address space, not just CICS

For calls *outbound* from WAS to external address space

Registration names *must be identical*

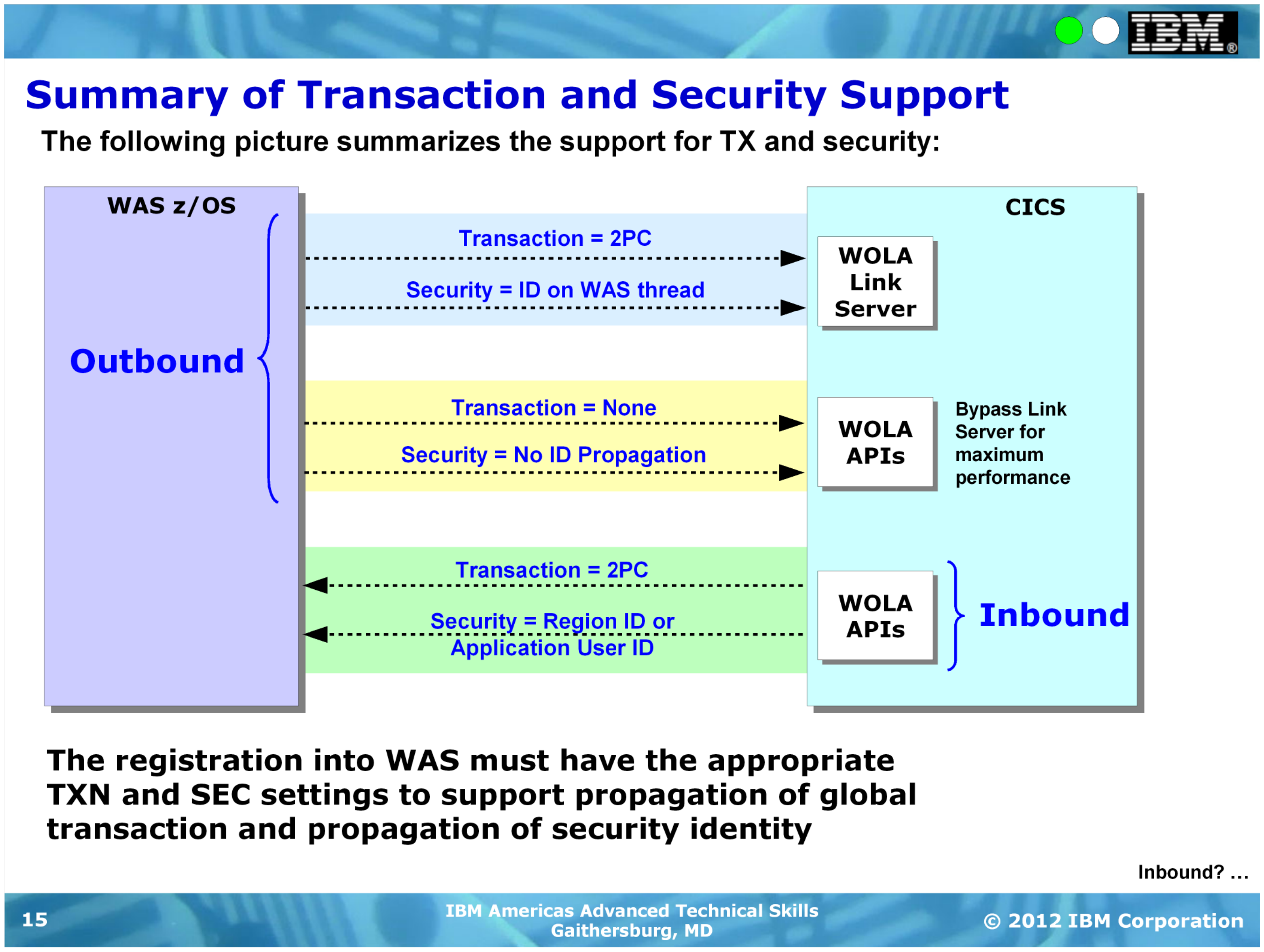
Targeted service must be present in address spaces participating in the work distribution

TX, Security summary ...

The WAS z/OS V8.0.0.1 fixpack also brought a round-robin function so two CICS regions registered in using the *same registration name* (yes, that is possible) may receive work coming from an application in WAS.

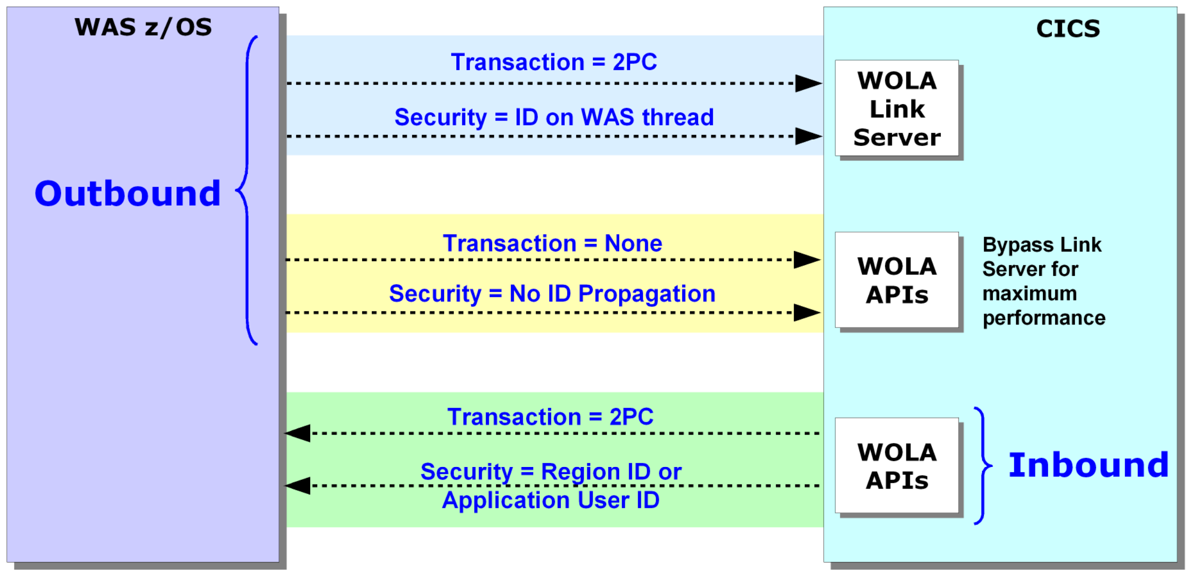
This is controlled by an environment variable (not a custom property on the connection pool or CF). A value of "1" means WAS is to honor the last registration into the WAS server, a value of "2" means to round-robin across the two like-named registrations.

Note: unlike with failover, this involves a single connection factory. And it's not necessary to code the registration name as a custom property on the CF. To the application it's as if there's one registration and one target CICS region. WAS knows better. It sees the two registrations and will round-robin across the two if you indicate that behavior with this new environment variable.



Summary of Transaction and Security Support

The following picture summarizes the support for TX and security:


WAS z/OS
Outbound
Transaction = 2PC
Security = ID on WAS thread
Transaction = None
Security = No ID Propagation
Transaction = 2PC
Security = Region ID or Application User ID
CICS
WOLA Link Server
WOLA APIs
Bypass Link Server for maximum performance
WOLA APIs
Inbound

The registration into WAS must have the appropriate TXN and SEC settings to support propagation of global transaction and propagation of security identity

Inbound? ...

This chart summarizes the transaction and security support provided by WOLA based on whether the call is outbound or inbound.

For outbound using the Link Server Task in the CICS region global transaction propagation is supported as well as propagation of the security identity on the WAS execution thread.

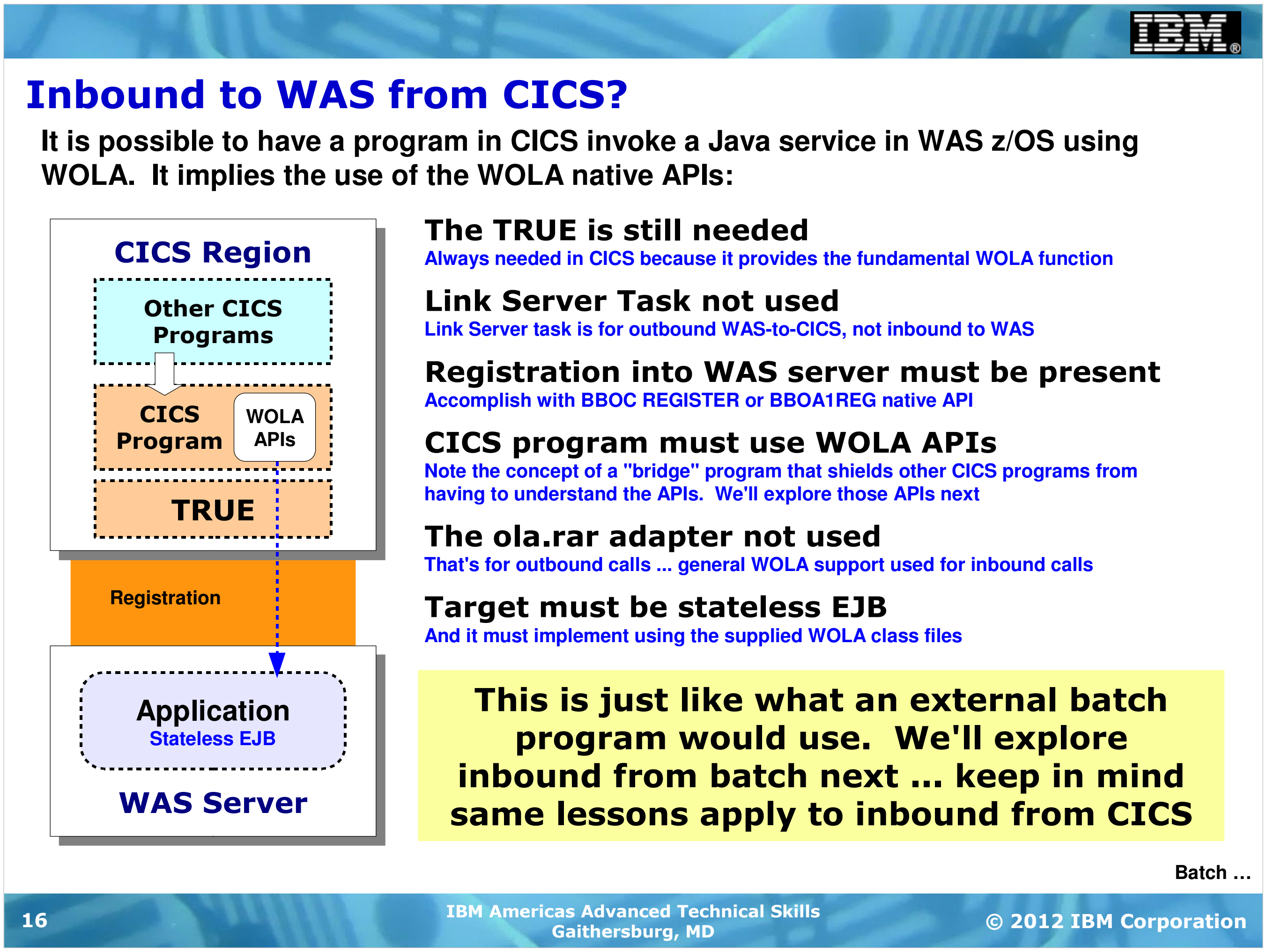
Note: the registration into the WAS server must indicate TXN=YES and SEC=YES for propagation to work.

If, however, you wish to bypass the use of the Link Server Task and code your CICS program to use the WOLA APIs directly, you lose both TXN and SEC propagation. Depending on your program requirements, it may be okay to run under the identity of the CICS region with transaction commit upon return.

Note: bypassing the Link Server Task for outbound calls to CICS is something one does when (a) the TXN and SEC requirements don't call for propagation, and (b) you are seeking the very best efficiency and throughput. The Link Server Task is very useful in that it shields your programs from WOLA knowledge, it supports TXN and SEC, and it makes using WOLA easier. But that comes with a degree of processing overhead. By coding directly to the APIs you may squeeze as much efficiency out of WOLA as possible for very high volume processing.

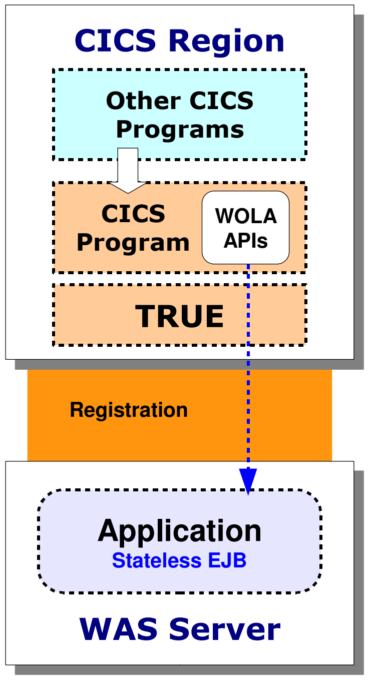
For inbound calls to WAS z/OS -- something we've not yet covered -- use of the WOLA APIs is required and propagation of security and transaction is possible. Again, the registration flags need to specify TXN=Y and SEC=Y for that aspect to funtion properly.

Let's take a closer look at the inbound support.



Inbound to WAS from CICS?

It is possible to have a program in CICS invoke a Java service in WAS z/OS using WOLA. It implies the use of the WOLA native APIs:



The TRUE is still needed
Always needed in CICS because it provides the fundamental WOLA function

Link Server Task not used
Link Server task is for outbound WAS-to-CICS, not inbound to WAS

Registration into WAS server must be present
Accomplish with BBOC REGISTER or BBOA1REG native API

CICS program must use WOLA APIs
Note the concept of a "bridge" program that shields other CICS programs from having to understand the APIs. We'll explore those APIs next

The ola.rar adapter not used
That's for outbound calls ... general WOLA support used for inbound calls

Target must be stateless EJB
And it must implement using the supplied WOLA class files

This is just like what an external batch program would use. We'll explore inbound from batch next ... keep in mind same lessons apply to inbound from CICS

Batch ...

16 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

WOLA is a bi-directional technology, supporting calls from WAS into CICS (outbound) as well as calls from CICS into WAS (inbound). Up to this point we've focused on the outbound model because that's one most think of when they think of WAS and CICS interacting with one another. But what about CICS into WAS invocation?

The keys to this are shown on the chart:

- The TRUE is needed to provide the low-level connectivity to use WOLA. The TRUE needs to be active at the time the program in CICS wishes to use WOLA.
- The BBO$ and BBO# link server task function is *not* used. That supports outbound calls that flow from WAS into CICS, but for the reverse -- CICS into WAS -- it is not used.
- The registration into the specific WAS server must be created. There are two ways this may be accomplished: using the BBOA1REG API in the program (more on this in the next section of this unit), or using the BBOC REGISTER command.
- The program in CICS must use the WOLA native APIs to interact with the application in WAS.

  Note: this does *not* mean every program in CICS must be changed. This means that at least one program needs to write to the WOLA APIs. Other CICS programs may DPL to the WOLA-enabled program for access to the WOLA function.
- The JCA adapter in WAS is not used. It may be present, but it's not required for inbound calls from CICS into WAS.
- Finally, the target of the inbound call must be a stateless EJB that makes use of the supplied WOLA class libraries for its home and remote interfaces. We'll see more detail about that coming up.

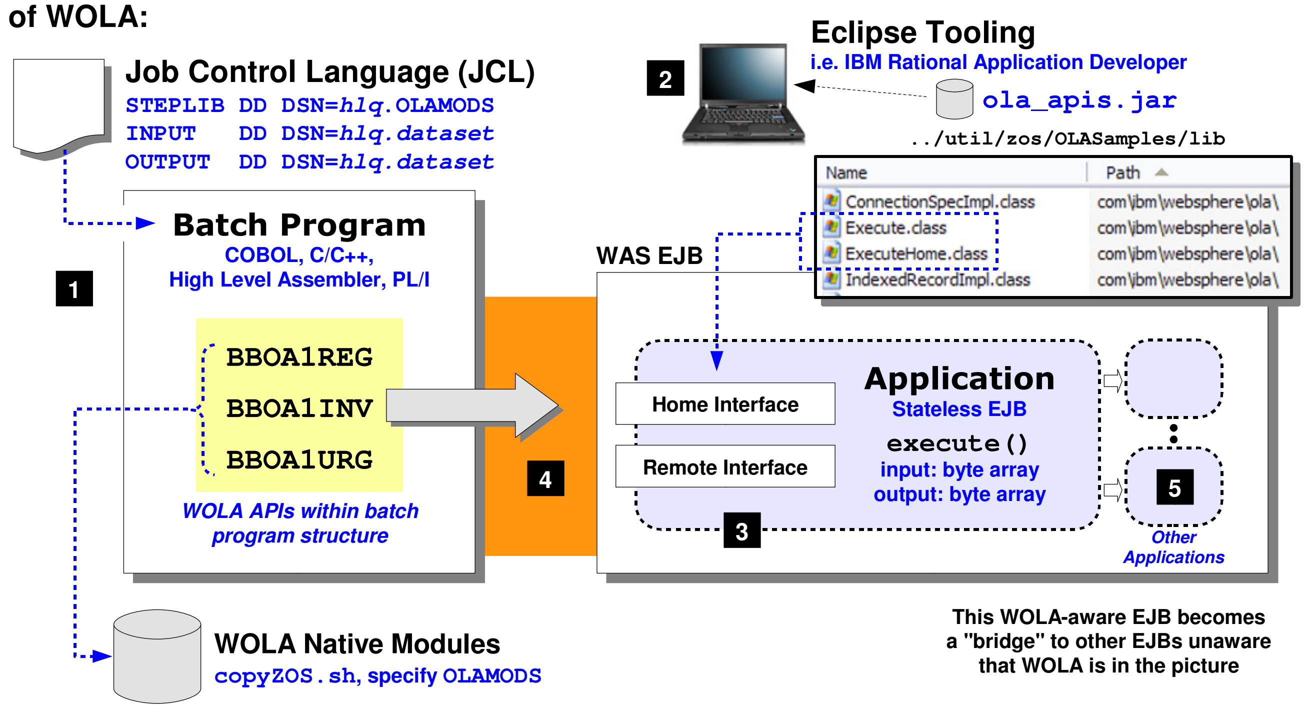
Rather than go into the details of the APIs for this inbound-from-CICS model we'll turn our attention now to batch programs using WOLA to reach into WAS and invoke EJBs. The model is the same as inbound from CICS with respect to the APIs and the programming requirements in WAS.

Inbound from Batch

Using the native APIs of WOLA

Essentials of Batch Program Use of WOLA

Relatively simple setup, but there is a bit more exposure to the programming interfaces of WOLA:



InfoCenter **cdat_olaapis, tdat_useola_in_step2**

API InfoCenter ...

Processing inbound to WAS z/OS from a batch program is a natural fit for WOLA -- batch programs by their nature perform repetitive operations on data records. Highly-efficient means of access inbound to WAS z/OS Java applications are limited. WOLA fits that bill nicely.

The chart above highlights the key requirements for doing this:

1. The batch program must use the WOLA native APIs to perform the WOLA calls into WAS. That means the JCL used to run the batch program must provide the program access to the native API module library. That module library is created using the `copyZOS.sh` shell script. The modules are copied out of the USS file system and into a pre-allocated `LIBRARY` data set. From there the batch program may `STEPLIB` to them.
2. The target application that will run in WAS z/OS must be developed and use the class library supplied in the `ola_apis.jar` file.
3. The target application must be a stateless EJB that implements the method `execute()` and takes as input a byte array and passes as output a byte array. The home and remote interfaces must be implemented using the classes found in the `ola_apis.jar` file. The application is then packaged and deployed into WAS as would be the case with any application.
4. As noted earlier, registration is always performed from the outside address space into the named WAS server. That's true for batch programs using WOLA as well. The BBOA1REG API (or BBGA1REG for 64-bit calls) is used to perform the registration. We'll look at the APIs in a bit more detail in the upcoming charts. For now the key is understanding that (a) registration must take place, and (b) the batch program does this using the register API. Then the batch program invokes the target application using the JNDI name of the deployed EJB as the "service" called by the WOLA invoke.
5. The EJB that serves as the target for the WOLA call may then turn and drive other EJBs in the the runtime. This is how you might use inbound WOLA to an ISV application where the ISV has not implemented WOLA classes in its program -- you write a small "bridge" (or "shim") program which receives the WOLA call, then it turns and drives the ISV application using the local interface.

The WOLA Native APIs InfoCenter Article

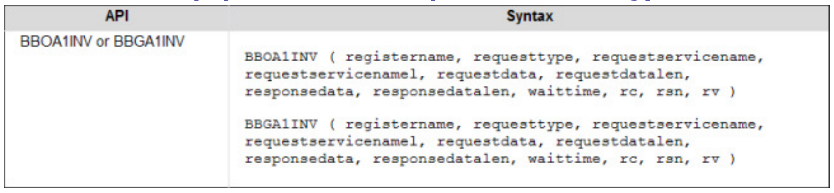
An incredibly useful InfoCenter article that details all 13 of the native APIs, including parameters and return code / reason code descriptions

13 APIs plus an internal link to JCA adapter APIs

- Register - BBOA1REG/BBGA1REG
- Unregister - BBOA1URG/BBGA1URG
- Connection Get - BBOA1CNG/BBGA1CNG
- Connection Release - BBOA1CNR/BBGA1CNR
- Send Request - BBOA1SRQ/BBGA1SRQ
- Send Response - BBOA1SRP/BBGA1SRP
- Send Response Exception - BBOA1SRX/BBGA1
- Receive Request Any - BBOA1RCA/BBGA1RCA
- Receive Request Specific - BBOA1RCS/BBGA1I
- Receive Response Length - BBOA1RCL/BBGA1
- Get Message Data - BBOA1GET/BBGA1GET
- Invoke - BBOA1INV/BBGA1INV
- Host Service - BBOA1SRV/BBGA1SRV
- JCA Adapter APIs

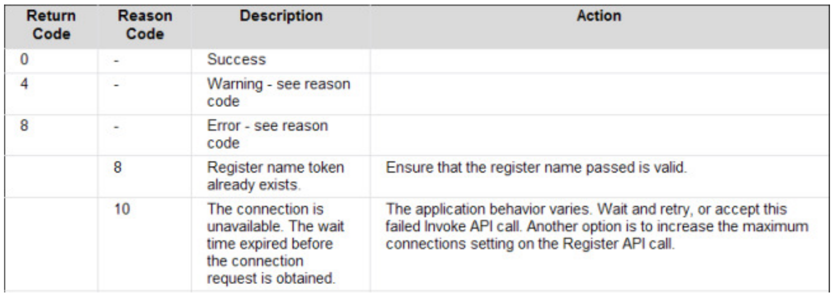
APIs that start with BBO* are 31-bit callable; BBG* are 64-bit callable

Parameter map (with full descriptions following)



API	Syntax
BBOA1INV or BBGA1INV	`BBOA1INV ( registername, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, responsedata, responsedatalen, waittime, rc, rsn, rv )` `BBGA1INV ( registername, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, responsedata, responsedatalen, waittime, rc, rsn, rv )`

Return Code / Reason Code descriptions for each API



Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	10	The connection is unavailable. The wait time expired before the connection request is obtained.	The application behavior varies. Wait and retry, or accept this failed Invoke API call. Another option is to increase the maximum connections setting on the Register API call.

A wonderful reference article, but it doesn't highlight how easy using the APIs can be …

InfoCenter `cdat_olaapis`

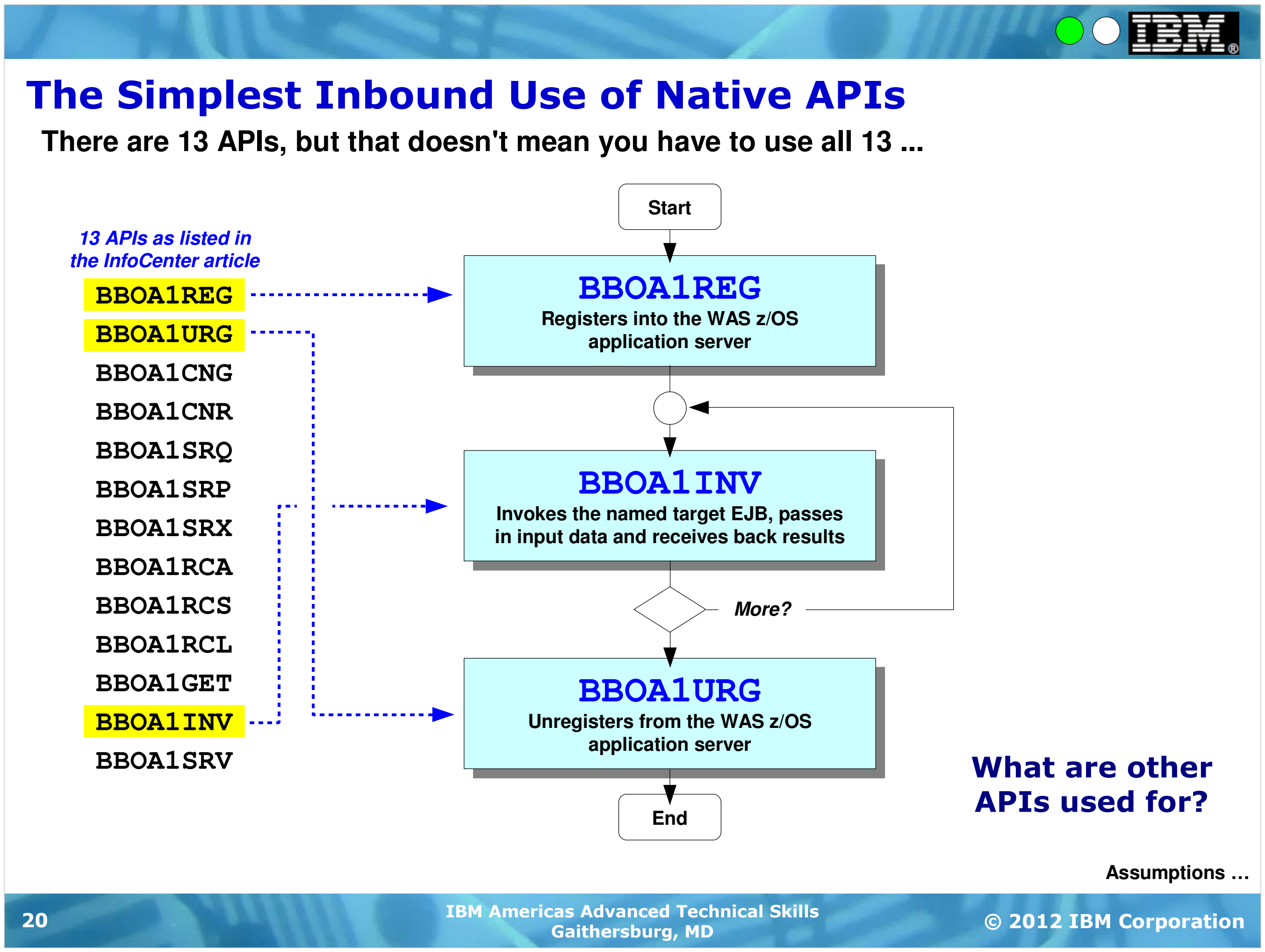
Simplest use …

19 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

The InfoCenter has an *excellent* reference article on the APIs located at keyword search `cdat_olaapis`. It provides a listing of all the APIs and provides a description of each as well as a parameter map and the return code and reason codes each may throw when operating.

Note: new with WAS V8 are 64-bit callable APIs for C/C++ programs. They carry the prefix `BBG*` rather than `BBO*`. APIs with `BBO*` are 31-bit callable APIs.

This InfoCenter article is the reference for APIs specifics, but it doesn't highlight how easy the APIs are to use. For example, as few as three APIs are needed to produce an inbound call program. 13 APIs are listed, but not all 13 are needed, depending on what you plan to do. So over the next few charts we'll explore how the APIs organize themselves into categories that make understanding their use a bit easier.



The Simplest Inbound Use of Native APIs

There are 13 APIs, but that doesn't mean you have to use all 13 ...

13 APIs as listed in the InfoCenter article
BBOA1REG
BBOA1URG
BBOA1CNG
BBOA1CNR
BBOA1SRQ
BBOA1SRP
BBOA1SRX
BBOA1RCA
BBOA1RCS
BBOA1RCL
BBOA1GET
BBOA1INV
BBOA1SRV
Start
BBOA1REG
Registers into the WAS z/OS application server
BBOA1INV
Invokes the named target EJB, passes in input data and receives back results
More?
BBOA1URG
Unregisters from the WAS z/OS application server
End

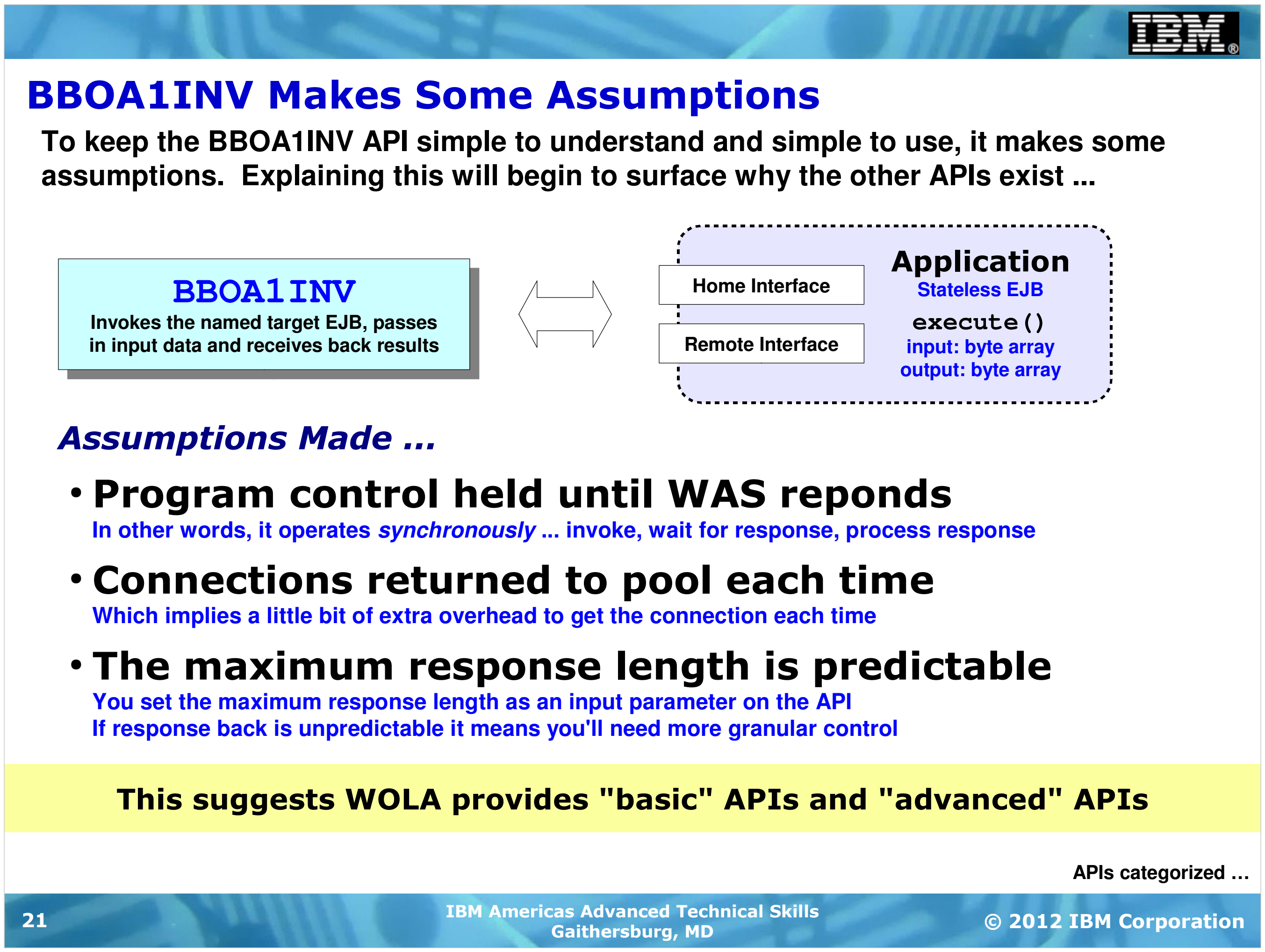
What are other APIs used for?

Assumptions ...

This is the simplest example of an inbound batch program invoking a target EJB in WAS z/OS. Of the 13 APIs listed in the InfoCenter article, only three are needed -- BBOA1REG, BBOA1INV and BBOA1URG.

Put a loop around BBOA1INV and you have an operation that may be repeated for as many times as the data records require. When you're done invoking the EJB you drop through to BBOA1URG to tear down the registration and exit the batch program.

This simplicity is made possible by the function of BBOA1INV making some assumptions about how it will operate. Let's explore those assumptions because they help us understand why there are 13 APIs. It has to do with how much control you wish to exercise in your program.



BBOA1INV Makes Some Assumptions

To keep the BBOA1INV API simple to understand and simple to use, it makes some assumptions. Explaining this will begin to surface why the other APIs exist ...

BBOA1INV
Invokes the named target EJB, passes in input data and receives back results

Home Interface
Remote Interface
Application
Stateless EJB
`execute()`
input: byte array
output: byte array

Assumptions Made ...

- **Program control held until WAS reponds**
  In other words, it operates *synchronously* ... invoke, wait for response, process response
- **Connections returned to pool each time**
  Which implies a little bit of extra overhead to get the connection each time
- **The maximum response length is predictable**
  You set the maximum response length as an input parameter on the API
  If response back is unpredictable it means you'll need more granular control

This suggests WOLA provides "basic" APIs and "advanced" APIs

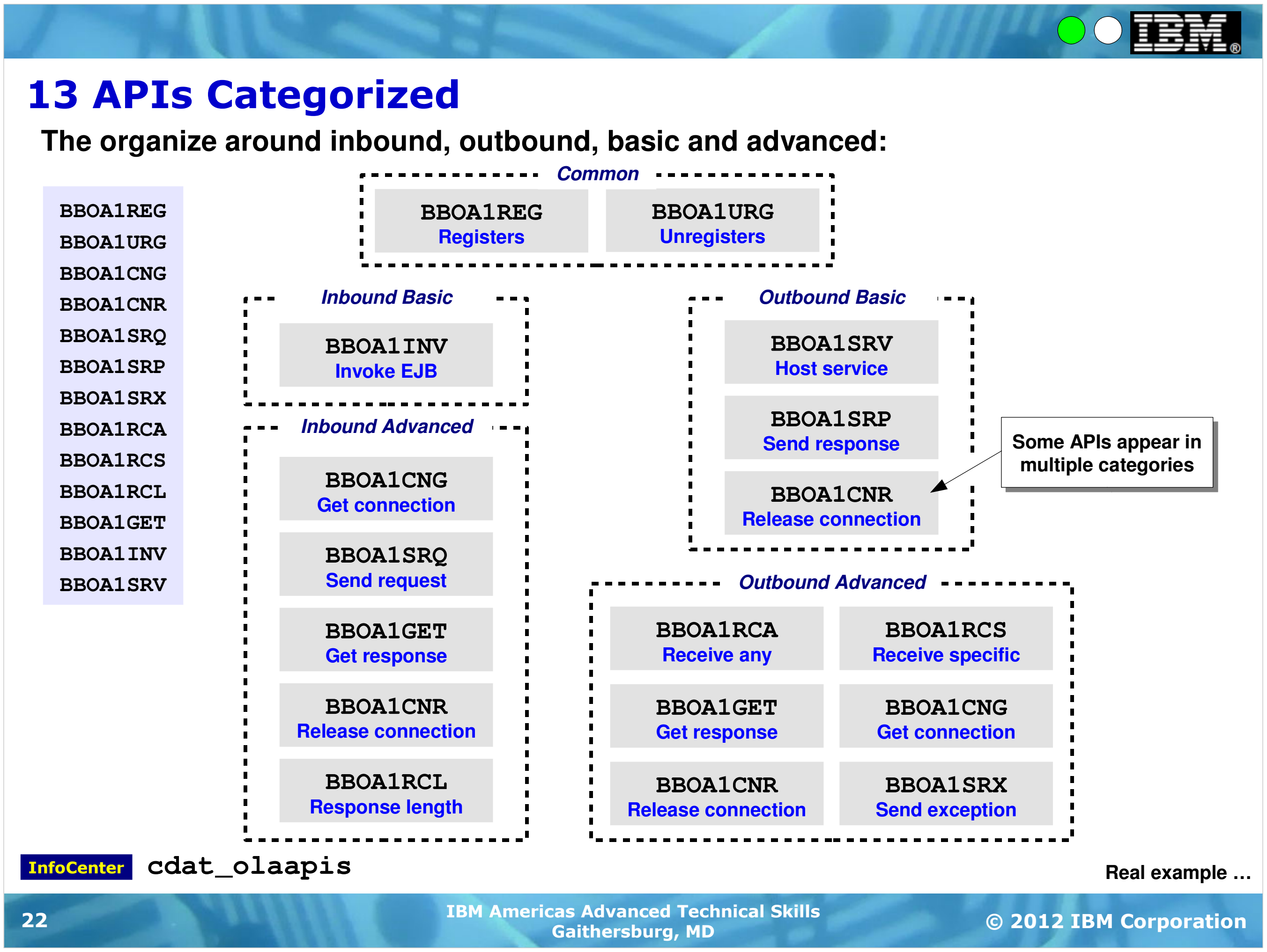
APIs categorized ...

21 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

To keep the use of `BBOA1INV` relatively simple the designers of WOLA chose to include some assumptions about programming behavior. Specifically:

- `BBOA1INV` operates *synchronously* ... that means when the batch program calls the API and the request flows over the WAS, program control is held from the batch program until WAS returns with the response. This makes things easier because your batch program does not need to check if a response is back -- you just invoke and when program control is returned you know a response is received. But, as you can well imagine, depending on how long it takes for the response to come back this may introduce a great deal of wait time in your batch program. This is why the other APIs permit operations to be *asynchronous* -- program control is returned immediately. Your batch program may then do other work. But you must then come back periodically to see if a response is back.
- Any given registration contains a pool of connections. How many is determined by parameters on the `BBOA1REG` API. BBOA1INV assumes that for each request a connection is retrieved from the pool, used, and returned. However, connections can be re-used, saving a little bit of processing each time. `BBOA1INV` does not re-use connections ... which is another reason why it's simple to use. It assumes each call of `BBOA1INV` implies getting a connection, using it, then returning it to the pool.
- Finally, `BBOA1INV` assumes that the maximum length of the response to come back is predictable. That is, you can predict with high certainty that the length will not exceed the value you specify as a parameter on the `BBOA1INV` API for return length. The other APIs provide a means of determining programmatically how long the response is, but BBOA1INV allows for maximum length and it processes that maximum length.

Our reason for highlighting these assumptions is to set the stage for the introduction of the other APIs and to begin to categorize the APIs around "basic" and "advanced," "inbound" and "outbound."



13 APIs Categorized

The organize around inbound, outbound, basic and advanced:

InfoCenter cdat_olaapis

Real example …

22 IBM Americas Advanced Technical Skills Gaithersburg, MD © 2012 IBM Corporation

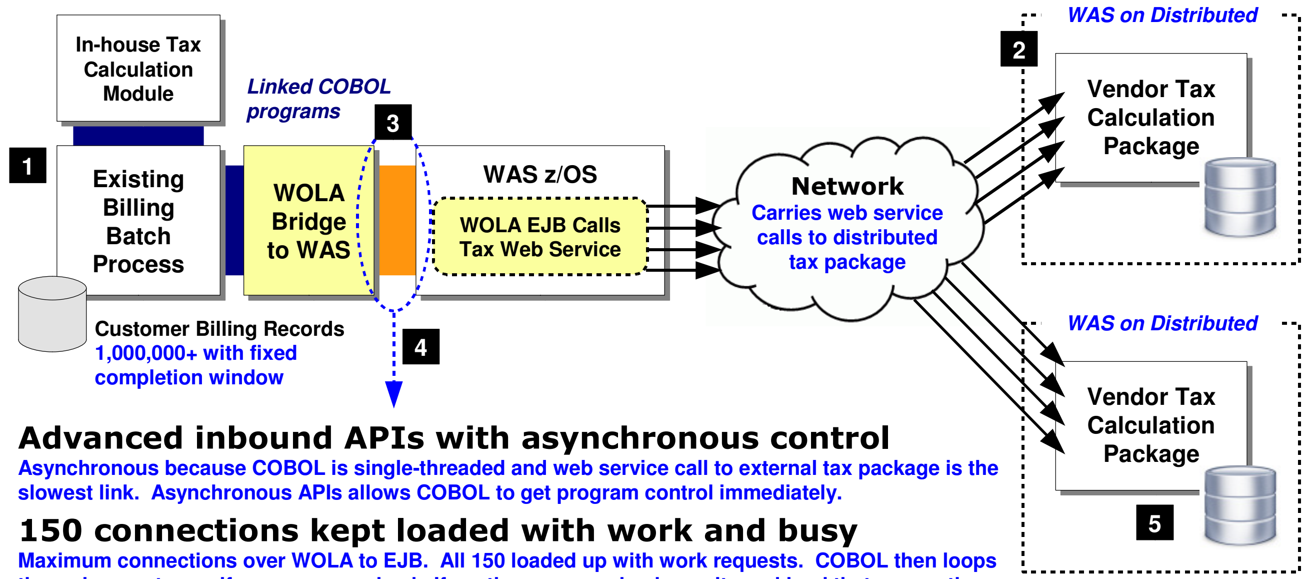
Here are the 13 APIs provided by WOLA categorized -- inbound vs. outbound, basic vs. advanced.

Note: we're not showing the 64-bit `BBG*` APIs here. They have the exact same function as the 31-bit `BBO*` APIs.

- *Common* - The register and unregister APIs are grouped under a "common" category because they apply in all cases.
- *Inbound Basic* -- only one API, `BBOA1INV`, and we just discussed the simplicity of this one API along with the assumptions made to provide that simplicity.
- *Inbound Advanced* -- these APIs introduce the notion of greater control over connection management and asynchronous operations. For example, `BBOA1CNG` is used to get a connection from the pool, `BBOA1CNR` is used to turn the connection back to the pool. You may re-use a connection multiple times if you wish, saving a little bit of processing through re-use. `BBOA1SRQ` is used to send a request, and it has a flag to be used asynchronously. If you run asynchronously, then you have to use either `BBOA1GET` or `BBOA1RCL` to see if a message has returned on the specific connection and pull it back.
- *Outbound Basic* -- calling outbound from WAS to a batch program requires pausing and thinking about what's required *before* the outbound call is made from WAS. The batch program would need to be started and registered into the WAS server, and the batch program would need to be a "listen" state. That's what the `BBOA1SRV` API does -- it "hosts a service." That service will sit in a wait state until invoked by the call from WAS. Then your program would use the `BBOA1SRP` to "send a response" back to WAS. These "basic" APIs operate synchronously, so like `BBOA1INV` they're relatively simple by virtue of the assumptions they make.
- *Outbound Advanced* -- these introduce the idea of asynchronous operations, specific connection management, and receiving calls for a specific service or any named service. Outbound advanced gets a little trickier to comprehend until you use the APIs in practice. The WP101490 API Primer covers the outbound advanced in a step-by-step manner.

A Real-Life Example of Inbound Batch Processing

This involves a COBOL batch program that invokes a vendor tax calculation application running on distributed WAS and accessed with web services:


In-house Tax Calculation Module
Linked COBOL programs
1
Existing Billing Batch Process
WOLA Bridge to WAS
3
WAS z/OS
WOLA EJB Calls Tax Web Service
Network
Carries web service calls to distributed tax package
Customer Billing Records
1,000,000+ with fixed completion window
4
WAS on Distributed
2
Vendor Tax Calculation Package
WAS on Distributed
Vendor Tax Calculation Package
5

Advanced inbound APIs with asynchronous control

Asynchronous because COBOL is single-threaded and web service call to external tax package is the slowest link. Asynchronous APIs allows COBOL to get program control immediately.

150 connections kept loaded with work and busy

Maximum connections over WOLA to EJB. All 150 loaded up with work requests. COBOL then loops through array to see if response received. If so, then process back results and load that connection with another request. Connections kept fully busy in this manner.

Multi-threaded Java then parallelized web service calls

WAS z/OS and WAS distributed are multi-threaded. Given sufficient processing capacity, the work requests from COBOL may then be handled in a parallel execution fashion.

Primer ...

What's represented here is a real-life use of WOLA. The numbered blocks correspond to the following notes:

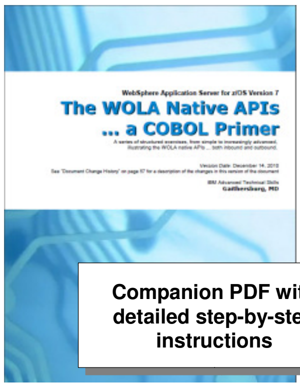
1. The existing batch process involved millions of customer billing records that needed to have tax calculated and applied. An in-house tax calculation module was linked to the batch program processing the billing records. The process has a requirement to complete within a certain time window.
2. Maintaining accurate tax calculation rules is a complicated process. Tax rates are based on many factors related to locality and calendar. Rather than trying to maintain an in-house rules engine, a vendor tax solution package was mandated. This vendor solution was implemented as a Java EE application, but certified to run only on distributed WAS. It had a web services interface.

Challenge: how to leverage existing billing batch process to utilize vendor tax package residing off-board with a web services interface and achieve the needed throughput (~3,000 records/second) to complete within the completion time window.

3. WAS z/OS was co-located with the batch process and a COBOL "bridge" program utilizing WOLA was written. An EJB written for WAS z/OS performed the outbound web services calls to the vendor package.
4. The real story is in how WOLA was utilized. The web service call was known to be the slow component in this cycle. If single-threaded synchronous WOLA processing was used the needed throughput would not be achieved. So the COBOL bridge program used the "advanced inbound" APIs to maintain 150 concurrent connections over WOLA into WAS z/OS. The APIs were used asynchronously so program control was returned to the COBOL program immediately. The COBOL program was designed to load all 150 connections with customer billing record requests, and to then cycle through the array of connections checking for which had come back completed. WAS z/OS and WAS on distributed are multi-threaded so the task was to (a) keep the flow of requests into WAS z/OS busy with the asynchronous WOLA calls, and (b) scale up the WAS z/OS and distributed WAS resources to achieve the needed throughput.
5. If needed, multiple instances of the vendor tax package can be deployed to further parallelize the environment. Benchmarking yielded the needed 3,000 records/second with one AIX server properly sized and tuned to handle the throughput. The WOLA connection easily kept up.

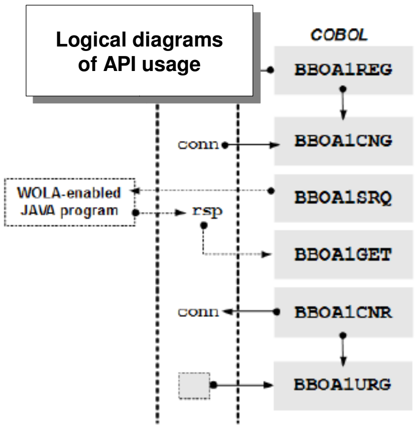
WP101490 Native API "Primer"

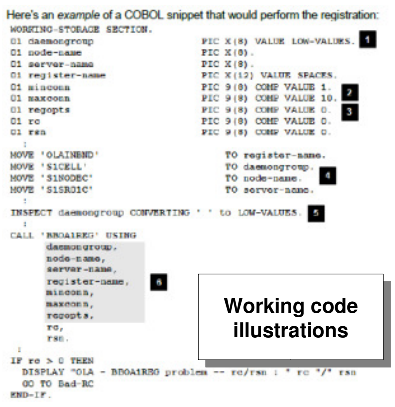
Provides a step-by-step introduction to the use of the native APIs with COBOL:



Companion PDF with detailed step-by-step instructions

Logical diagrams of API usage

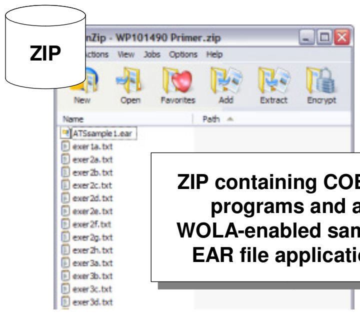




Here's an *example* of a COBOL snippet that would perform the registration:

```
WORKING-STORAGE SECTION.
01 daemongroup                PIC X(8) VALUE LOW-VALUES.  1
01 node-name                  PIC X(8).
01 server-name                PIC X(8).
01 register-name              PIC X(12) VALUE SPACES.
01 mincon                     PIC 9(8) COMP VALUE 1.     2
01 maxconn                    PIC 9(8) COMP VALUE 10.
01 regopts                    PIC 9(8) COMP VALUE 0.     3
01 rc                         PIC 9(8) COMP VALUE 0.
01 rsn                        PIC 9(8) COMP VALUE 0.
  :
MOVE 'OLAINBND'                   TO register-name.
MOVE 'S1CELL'                     TO daemongroup.   4
MOVE 'S1NODEC'                    TO node-name.
MOVE 'S1SR01C'                    TO server-name.
  :
INSPECT daemongroup CONVERTING ' ' to LOW-VALUES.  5
  :
CALL 'BBOA1REG' USING
      daemongroup,
      node-name,
      server-name,
      register-name,     6
      minconn,
      maxconn,
      regopts,
      rc,
      rsn.
 :
IF rc > 0 THEN
  DISPLAY "OLA - BBOA1REG problem -- rc/rsn : " rc "/" rsn
  GO TO Bad-RC
END-IF.
```

Working code illustrations



ZIP containing COBOL programs and a WOLA-enabled sample EAR file application

When you're ready to begin using the native APIs, this "Primer" will assist you in understanding how the APIs are used

8.0.0.1 ...

Approaching the WOLA APIs may seem like a daunting challenge, but with a little help it's really quite easy. The WP101490 Techdoc contains a step-by-step "Primer" document that will assist in this learning curve.

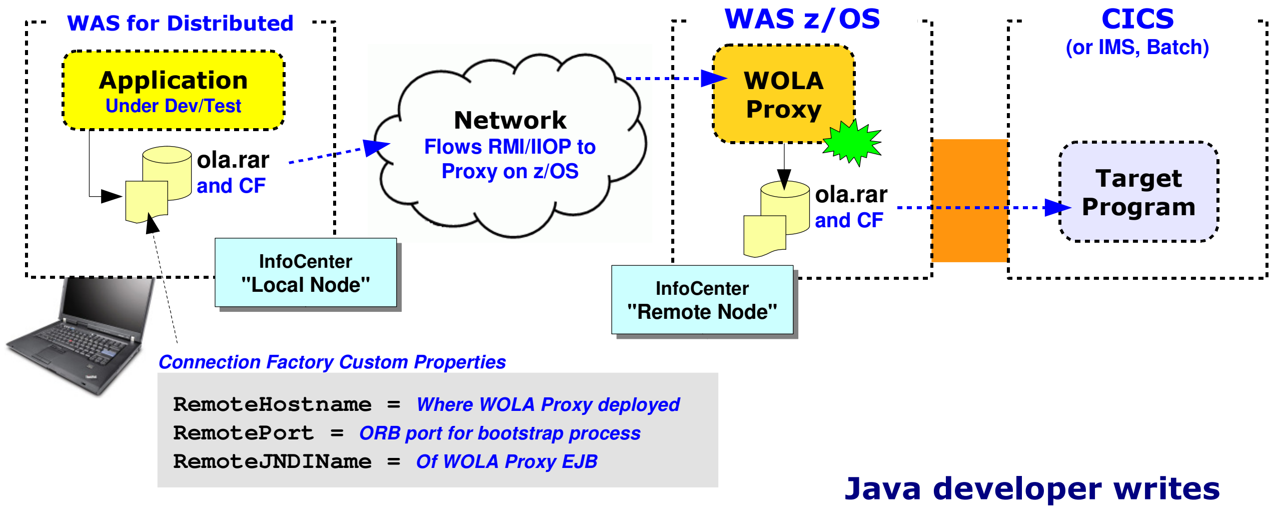
The Primer consists of a series of guided exercises (with supplied code samples) to exercise the APIs from simplest to increasing complex usage scenarios.

New in V8.0.0.1

"Development Mode" using the Proxy Application

Development Mode - *Outbound* Applications

The focus here is on developing and testing WOLA outbound applications without the developer needing direct access to a z/OS system



```
RemoteHostname  =  Where WOLA Proxy deployed
RemotePort  =  ORB port for bootstrap process
RemoteJNDIName  =  Of WOLA Proxy EJB
```

Limitations:

- Can not participate in global transaction 2PC
- Can not assert distributed WAS thread ID up to z/OS.

Java developer writes application to CCI in the WOLA JCA resource adapter just as if the application was deployed on WAS z/OS

InfoCenter `cdat_devmode_overview`

Inbound ...

One of the challenges Java developers face when working on WOLA-enabled applications is the ability to do their development and test on their workstations. WOLA is a z/OS technology and works cross-memory to address spaces on the same LPAR. So how can applications be developed and tested without having to deploy the application to a WAS z/OS instance each time? By providing a "proxy" function with WOLA so the application can be tested on the development workstation and its requests appear to be talking using WOLA.

This function came into being in the 8.0.0.1 fixpack. It's referred to as "Development Mode," and it takes two forms -- *outbound* development mode (shown above), and *inbound* development mode (next chart).

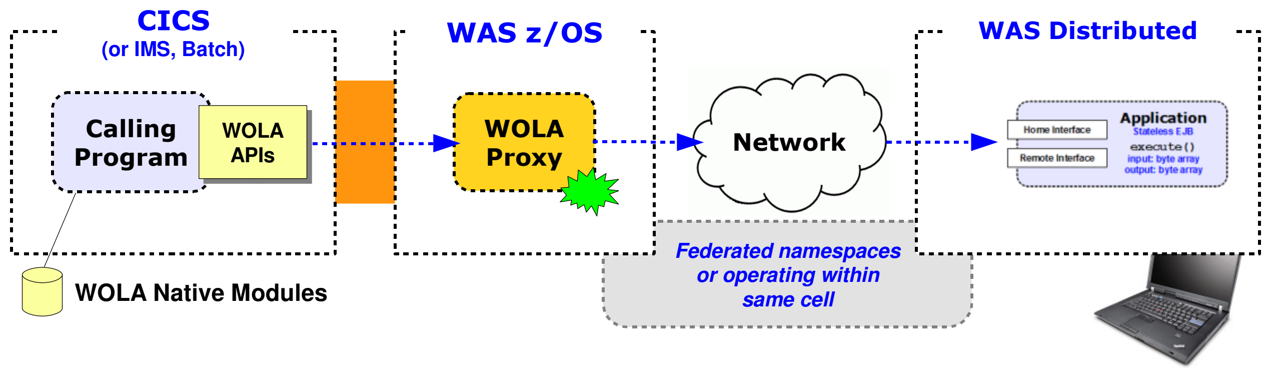
For outbound development mode the objective is to provide an ability for developers coding Java applications that use the WOLA JCA resource adapter the ability to test their code against a real target z/OS address space ... but without requiring their application be deployed to WAS z/OS each time.

New with 8.0.0.1 is a Proxy application that deploys on WAS z/OS that will use WOLA against a target external address space such as CICS. The Proxy application stands ready to accept calls from development and test machines. Those workstations use the 8.0.0.1 WOLA JCA resource adapter -- which may be downloaded and installed in WAS on any platform -- to forward application requests up to the proxy application. To the application on the development workstation (which the InfoCenter refers to as the "local node") it's just the Common Client Interface (CCI) as implemented in a standard JCA resource adapter. The key is the definition for the connection factory for the WOLA JCA resource adapter ... it now accepts custom properties that allows it to bootstrap into a WAS z/OS environment and communicate with the Proxy application using RMI/IIOP.

In a nutshell ... the application under development/test on the workstation calls the CCI interfaces of the JCA resource adapter. That resource adapter then uses RMI/IIOP to forward the requests up to the Proxy application on WAS z/OS. The Proxy application on WAS z/OS then honors the calls and turns and drive actual WOLA against the the target external address space.

Development Mode - *Inbound* Applications

Let's take the reverse ... the case where you wish a native z/OS program to make an inbound call to a target EJB running in WAS. Can EJB be on WAS distributed? Yes ...



WOLA API developer writes as if target EJB is in the WOLA-attached WAS z/OS server

One parameter difference - `requesttype` on `BBOA1INV` or `BBOA1SRQ` set to "2" (for remote EJB request) rather than "1"

EJB Developer develops stateless EJB with WOLA class libraries as if deployed on z/OS

InfoCenter `cdat_ola_remotequest`

The new 8.0.0.1 development mode supports the reverse as well -- applications that make use of the WOLA native APIs to call a target stateless EJB. The target EJB may now reside on a development WAS distributed machine away from z/OS.

The Proxy application supplied by WOLA in 8.0.0.1 is still required to be in a WAS z/OS server that can be reached using "real" WOLA from the external address space (CICS, Batch, IMS) into WAS. On the development machine the EJB simply needs to be active in a WAS distributed server.

The WOLA Proxy application needs to perform a JNDI lookup of the "service," which is the JNDI name of the target EJB. For it to do that it'll need knowledge of the JNDI namespace of the remote development workstation WAS. That may be accomplished either by federating the namespaces of the two WAS cells (information on doing this is available in the InfoCenter article cited on the chart), or by having the two environments (z/OS and distributed) be part of the same WAS cell.

In any event the native API develop has an environment where the target EJB is off on some development workstation. The only programming change needed is a single parameter on the `BBOA1INV` or `BBOA1SRQ` APIs -- setting requesttype to a value of `2` to tell the Proxy application to honor the proxy and forward on. The developer of the EJB does not make any accomodations for this other than to follow the normal rules for developing a target EJB for WOLA -- that is, it must implement the method `execute()` and it must use the supplied WOLA class libraries for its home and remote interfaces.

End of Unit