WP101490

WebSphere Optimized Local Adapters for WAS z/OS

Overview and Usage

ibm.com/support/techdocs

© 2012 IBM Corporation

Contents

The flow of this document is organized around the following main topics:

Setting the Stage

Basic Components

Fundamental Concepts

Specific External Address Space Usage
- **WOLA and Batch**
- **WOLA and CICS**
- **WOLA and IMS**

Finer Details

Wrap-Up

Other WP101490 documents ...

 © 2010 IBM Corporation

This is the flow of the material in this document. It is designed to take you into the topic of WOLA without overwhelming you with too many details or exceptions at the start. WOLA has a lot of details associated with it, but underlying those details are some fairly easy-to-follow concepts. That's what we'll do in this presentation, going from high level to increasingly deeper level.

© IBM Corporation 2010

Other Sources of Information

From the WP101490 Techdoc:

Technical Brochure

A "glossy brochure" format that gives a high-level of WOLA and answers a few anticipated questions.

Planning Guide

Providing specific information and InfoCenter pointers based on the type of WOLA usage you intend.

Native APIs Primer

A guided instructional exploration of the WOLA native APIs with COBOL

On Wikipedia:

`http://en.wikipedia.org/wiki/WebSphere_Optimized_Local_Adapters`

Or search WOLA and "disambiguate"

On YouTube:

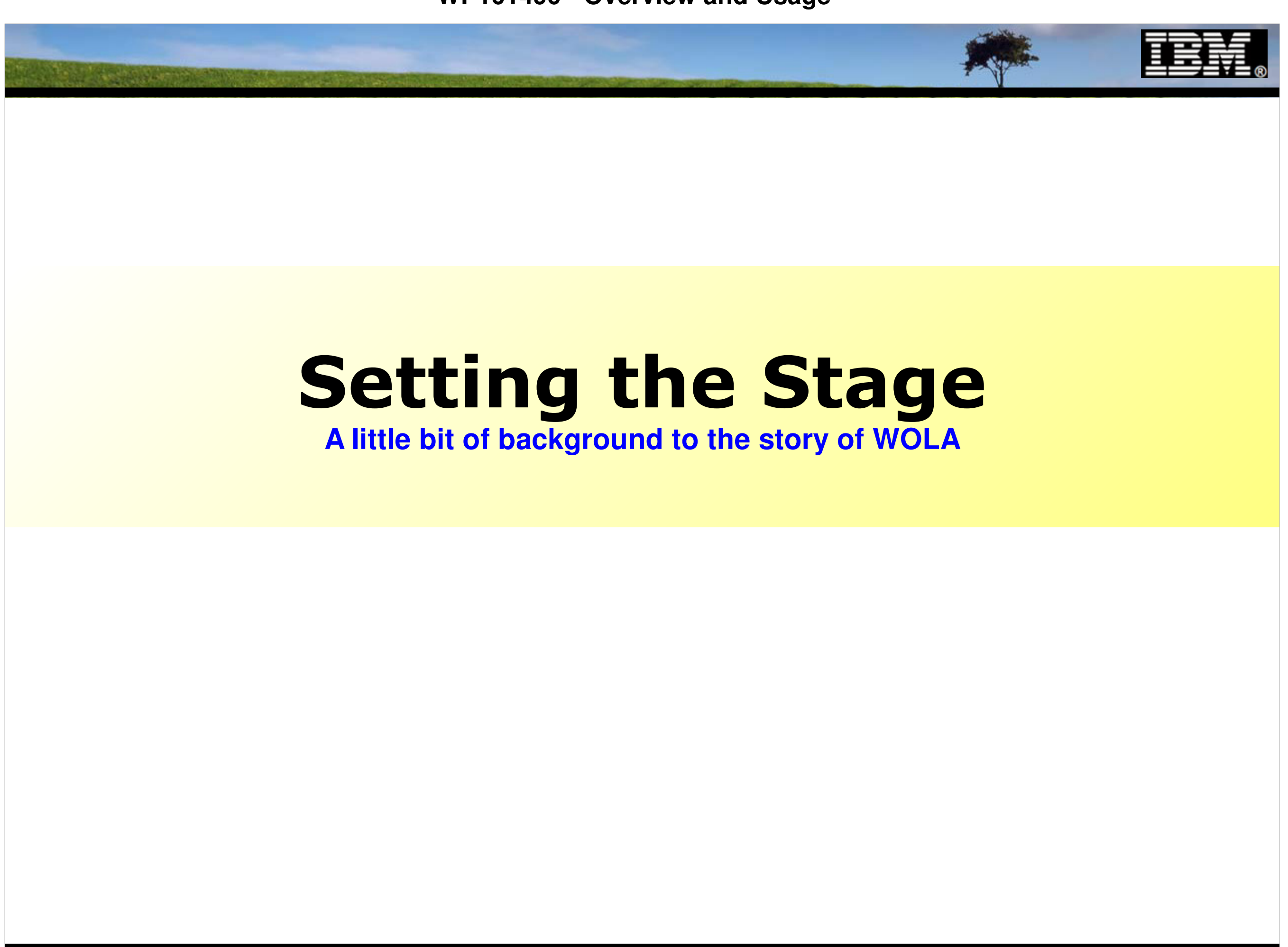
`http://www.youtube.com/results?search_query=WASOLA1&aq=f`

Setting the stage ...

 © 2010 IBM Corporation

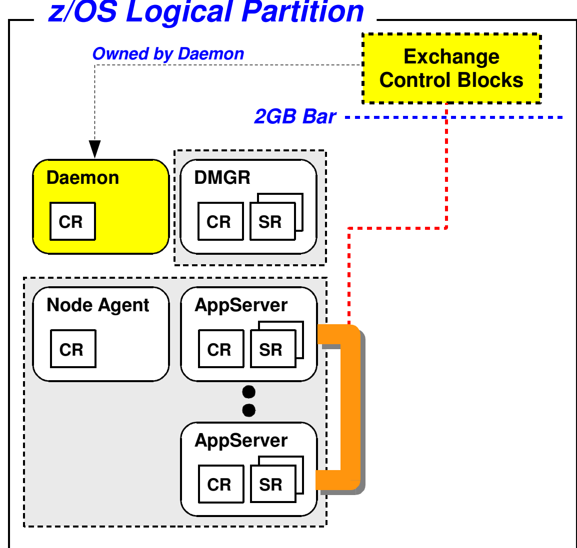
This presentation is not the only source of information about WOLA. The WP101490 Techdoc, from which this presentation came, also has two other documents you may find worthwhile.

© IBM Corporation 2010


IBM
Setting the Stage
A little bit of background to the story of WOLA
4
IBM Advanced Technical Skills
© 2010 IBM Corporation

In The Beginning Was "Local Comm"

This has been around since the early days of WAS on z/OS. It's a way to bypass the TCP/IP stack for internal IIOP calls between servers on the same LPAR:



If an IIOP call is made and WAS z/OS sees it's on the same LPAR, then this is automatically done

The Daemon server plays a key role in this; it owns the above-the-bar space used shared space and does the inter-address switching

It's very fast with very low overhead

The initial motivation ...

5 IBM Advanced Technical Skills © 2010 IBM Corporation

To better understand what WOLA is we need to step back in time and take a look at a technology that's been used by WAS z/OS since the very early days of the product. That function is something called "Local Communications," or "Local Comm" for short. It is a way for WAS z/OS to recognize that an IIOP call is targeted for an object in the same z/OS operating system and to use the z/OS cross-memory services rather than invoking the TCP/IP stack to perform the call.

Note: z/OS also has something called "Fast Local Sockets" for TCP calls within the same z/OS operating system. That's a way of using only TCP (and not IP) for faster "local" socket calls. But Local Comm is faster still. It does not use TCP at all.

What the picture above is representing is a somewhat high-level schematic of how Local Comm is implemented. The Daemon server of WAS z/OS owns a chunk of above the bar shared memory that's used to hold control blocks used for the cross memory transfer. That shared memory is not in the Daemon address space, it is simply owned by it.

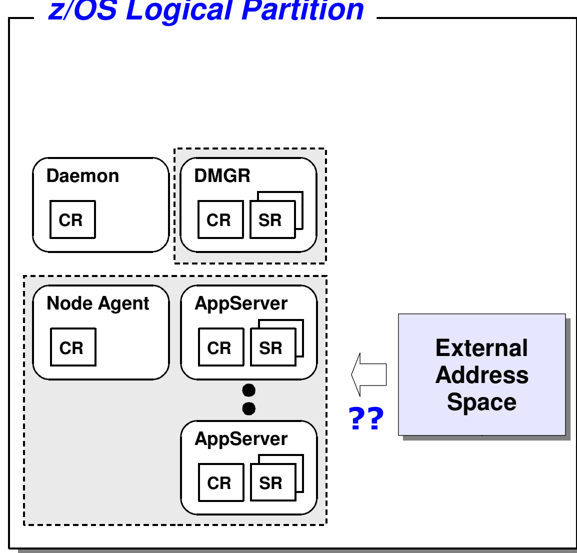
When an IIOP call is initiated and targeted for an object in the same cell on the same z/OS operating system instance (we'll use LPAR to mean that from this point forward), the Local Comm function copies t the control block area the pointer to the memory location. Then it lets the target process know the pointer reference and it authorizes the fetching of the message from that location. Since it's across address spaces, and z/OS in general protects at the address space level, something with sufficient authority needs to control that. That something is the Daemon server, which is why it owns the shared space.

In this picture the thick orange bar represents the cross-memory transfer, with the dotted red line to the shared space yellow block representing the ownership and control of the Daemon.

Got it? That's Local Comm. And as mentioned, it's been around for a long time.

The Motivation Behind WOLA

It *started out* as a way to allow program access *into* WAS for high transaction rate batch programs. Other solutions existed, but they all had limitations:



Inbound to WAS?

As more and more solutions are built based on Java EE, there is a growing desire to access them by batch, CICS and IMS programs

MQ or Web Services?

Both are very good technologies and have their role. But for very high throughput and low overhead, each has their drawbacks.

Something else was needed ... something *very fast* with as *little overhead* per exchange as possible

The birth of WOLA ...

What got the developers thinking about something like WOLA? Initially it was a desire to provide a way from outside WAS in to invoke business logic and services located in WAS z/OS. Particularly batch processes that need to operate quickly, efficiently and with a minimum of overhead.

There are ways to accomplish this -- MQ, web services -- and those are very good technologies. But each has its share of overhead and limitations.

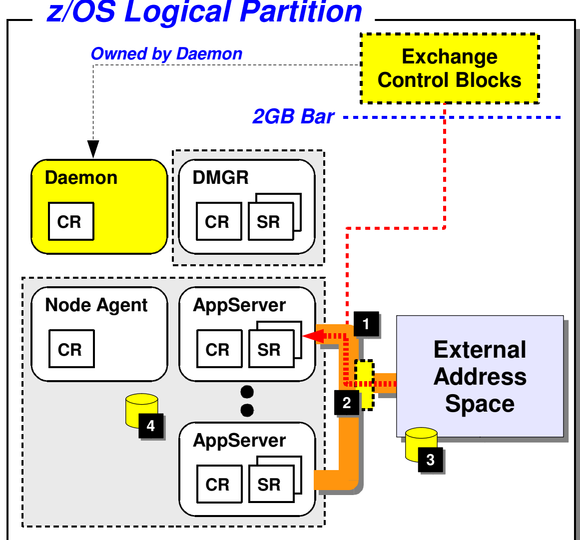
The developers got to thinking about this challenge and remembered the Local Comm function inside WAS z/OS. That was the initial spark ... perhaps they could "piggyback" on Local Comm and provide a way into WAS.

And thus WOLA was born ...

© IBM Corporation 2010

Answer: Externalize the Local Comm Function

The Local Comm function was there. It just needed interface modules so external address spaces could access it:



WOLA was born

1. **Existing Local Comm exploited**
2. **Externalization routines written**
3. **Programming APIs for external address spaces provided**
4. **Standard JCA adapter for the WAS server provided**

Just Inbound? No! This is a bi-directional technology. Outside into WAS, and WAS out to external address space

What external address spaces? ...

What the developers did was externalize the Local Comm function so it was accessible by address spaces outside the WAS z/OS cell. They did this with four basic components:

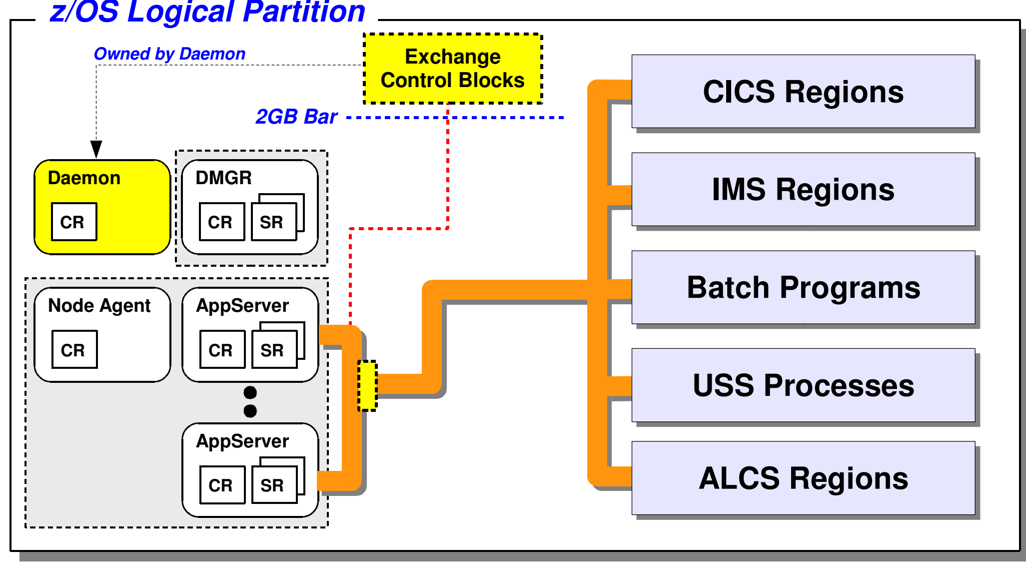
1. By using the existing Local Comm ... this is what we spoke of on the previous chart. This is *not* a reinvention of the wheel. This is the exploitation of existing function ... extended out.
2. By writing some very low-level externalization modules that lets the Local Comm function know about and understand there's more to its world than just itself.
3. A set of programming APIs for programs in external address spaces so they could access this Local Comm infrastructure and target the business logic in WAS z/OS in a systematic and documented way.
4. A standard Java Connector Architecture (JCA) resource adapter that deploys into WAS z/OS so the Java programs there could have a standard interface -- the Common Client Interface (CCI) -- to interact with.

The developers understood full well that if they stopped with just inbound-to-WAS support the very next question would be "What about the other direction?" So they kept banging on their keyboards and made this technology bi-directional; that is, the external program can call into WAS z/OS, and Java programs can call out to external programs.

External programs where? That's next ...

What External Address Spaces Supported?

The picture looks like this ... recently enhanced with IMS in the 7.0.0.12 fixpack:



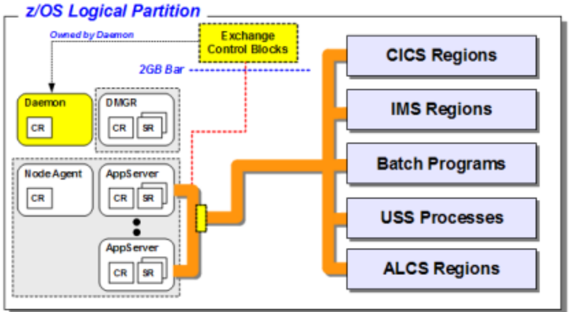
Let's slow down ...

This picture shows a very abstract representation of what we were talking about earlier, expanded to include the different types of external address space structures WOLA can work with. We'll spend the rest of this presentation unraveling the details of CICS, IMS and Batch Programs.

Note: the WP101490 Techdoc has a brochure on ALCS if you're interested. ALCS is a high performing transaction server environment used in the airline and reservations industry. It gets its efficiency by keeping a clear focus on being a fast transaction processor and not trying to be something for everybody. We won't cover USS processes in this presentation, but the concepts we'll cover for batch apply to USS as well.

We've Only Just Begun ...

The previous pictures set the basic framework in place, but there are many details left to explore



This is a somewhat abstract picture.

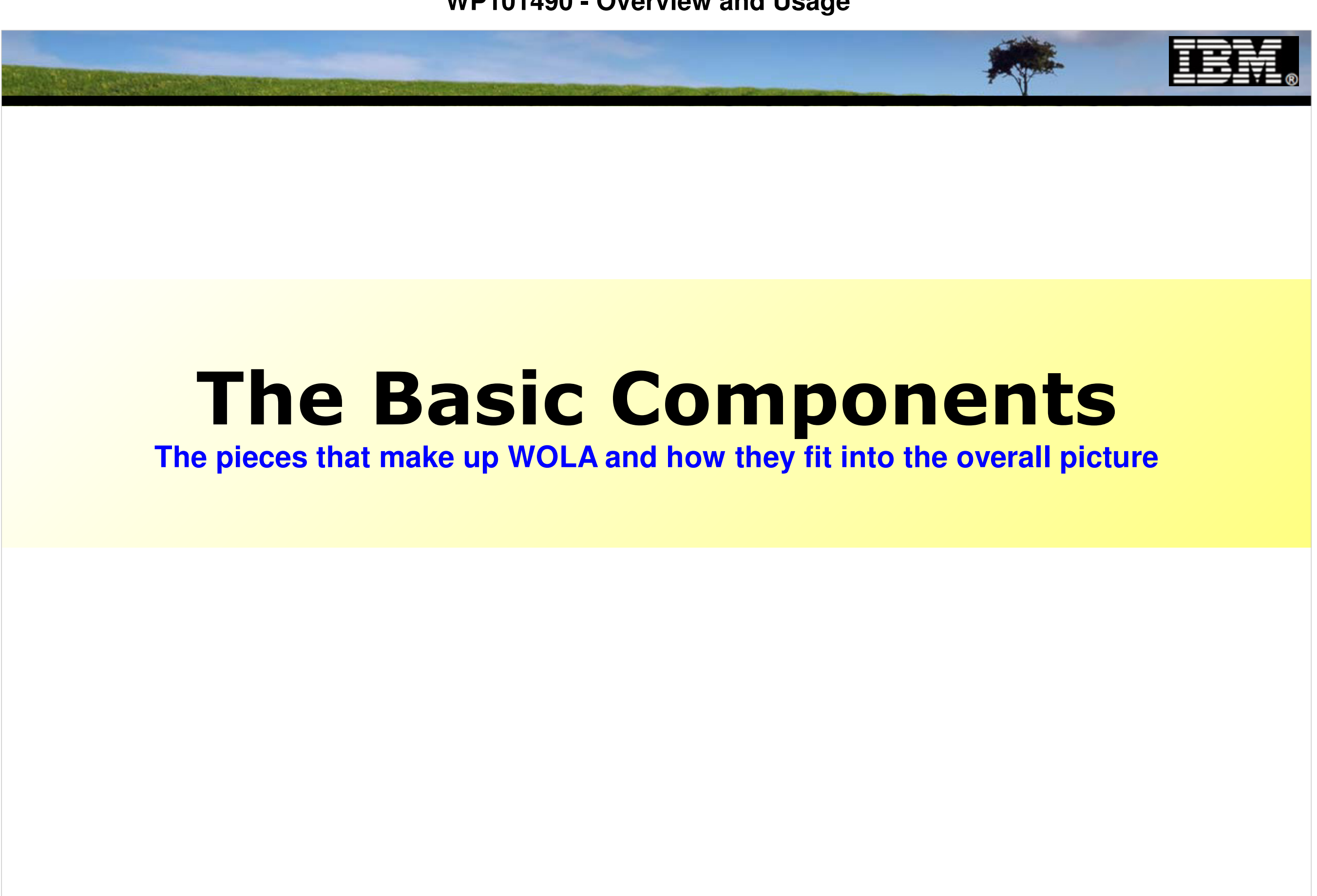
Key concepts are correct but many details have been omitted

It's important to keep in mind that the different external address spaces have different characteristics, and thus different WOLA considerations

With that, let's start digging under the covers ...

9 IBM Advanced Technical Skills © 2010 IBM Corporation

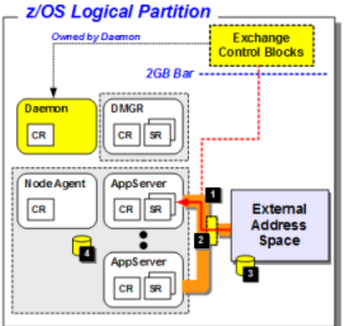
There is likely a strong urge on your part to start asking questions about the specifics of how this works. We will start to do just that. Please be a bit patient with us ... to tell this story in a systematic way requires a few setup charts, then we get into the details of each environment.



© IBM Corporation 2010

Key Components

The earlier picture gave a hint to some of the components. We'll start to review them a bit more closely here



WOLA was born

1. Existing Local Comm exploited
2. Externalization routines written
3. Programming APIs for external address spaces provided
4. Standard JCA adapter for the WAS server provided

Proper level of WAS z/OS

- Function made available first in 7.0.0.4
- Recently enhanced to include IMS and other functionality in 7.0.0.12

WOLA support enabled in the WAS z/OS cell

- Shell script to create symlinks to the WOLA files
- Single WAS environment variable to turn function on

WOLA JCA adapter installed in the WAS z/OS nodes

- Simple JCA adapter installation like any other

External WOLA modules copied out to library

- A supplied shell script will do this for you
- This is how you make function available to others

Enablement work done in external address space

- For batch program it's as simple as STEPLIB
- For CICS and IMS there's a few easy Sysprog tasks to do

Sample code used to validate the environment

- Sample EAR file provides the WAS-side code
- Sample COBOL and C/C++ code provides the external

There's very little here that's complex for the Sysprog. Fairly straightforward stuff.

The WAS z/OS InfoCenter has very good documentation on this

Key concepts ...

11 IBM Advanced Technical Skills © 2010 IBM Corporation

This chart is in the deck to give you a sense for the various setup work needed to get this to work. We're not going to dwell on this things too much because, for the most part, they are fairly straightforward things. The InfoCenter has very good step-by-step documentation on all this.

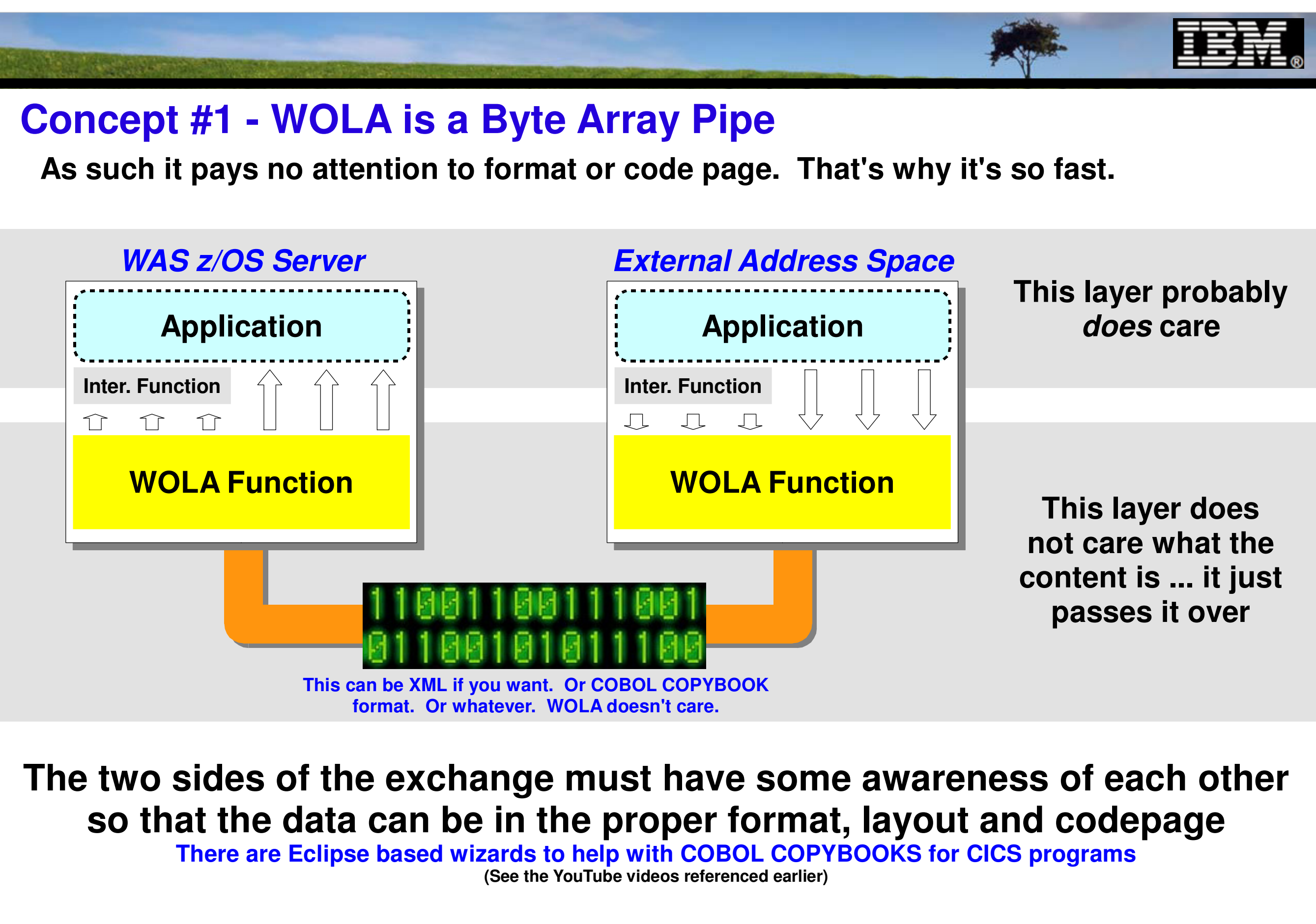
What we're going to do is focus on *using* the functionality once it's been enabled.

A Few Fundamental Concepts

These are absolutely essential to understanding the details that follow

© 2010 IBM Corporation

© IBM Corporation 2010



The first key concept we wish to establish is that WOLA is itself a very low-level byte array pipe. By that we mean that the WOLA function itself pays no attention to the contents of what's being passed back and forth. It doesn't care about data layout and it doesn't care about code pages. It slings byte arrays back and forth very quickly.

But that's not to say that the application layer can ignore those considerations. In fact, very likely they can not ignore them. So at some point there'll have to be some consideration of these issues at the application layer.

One such case would be CICS programs that work against a COBOL COPYBOOK. There are Rational wizards to import a COPYBOOK structure and format up the Java-side handling of that. That allows the Java program to format up the data in such a way the COBOL program in CICS can properly interpret it. (Early on the "Other Sources of Information" page we referenced a YouTube link ... Jim Mulvey, the lead architect of WOLA, discusses some of those techniques.)

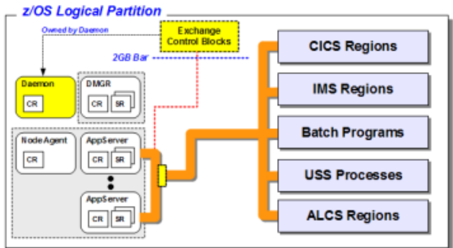
The point here is that WOLA itself gets a blob of bytes and shuffles them quickly over the WOLA pipe. It does that very well. And it does that very quickly by not getting involved in parsing and conversions.

You'll notice the little gray box "Inter. Function." By that we mean it's possible to place an intermediary function between the low-level WOLA structure and the end-using application. We'll see that take shape in the CICS support in the form of the "Link Server Task." Also, you could code up a simple "bridge EJB" that interfaces with WOLA and does whatever tweaking necessary before passing the data to your business function program. That's what we mean by "intermediary function."

Concept #2 - WOLA is Address Space to Address Space

This is a very low-level mechanism between address spaces:

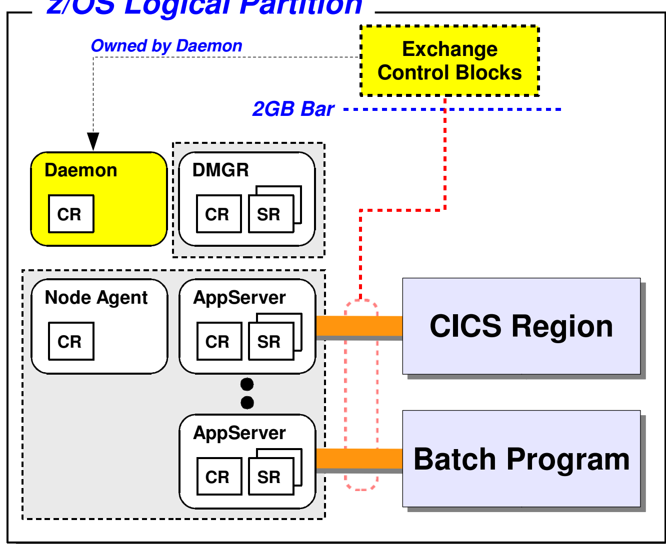
This picture from earlier was a bit misleading



To exchange with an application in a server, it is required to register to *that specific server*

WOLA does not provide general access to the WAS cell, it provides very low-level access to the specific server and applications in that server

*This is more technically correct**



Multiple registrations permitted, to the same server or different servers

* Strictly speaking the connection is to the WAS server controller region

"Registration" ...

14 IBM Advanced Technical Skills © 2010 IBM Corporation

The second key concept we wish to establish is that WOLA is an address space to address space communication link. That's where it gets its speed and efficiency: by being a very low-level exchange mechanism and *not* a general purpose routing function. Other technologies exist for flexible routing from on platform or off. WOLA is very definitely meant to be from an external address space to *a specific WAS z/OS server*.

Note: more specifically, a specific WAS server *controller* region. Servant regions know nothing of WOLA.

Our preliminary picture seemed to suggest a more generic connectivity. The picture on the right is more precise. (That reference to "registrations" will be explained in a bit ... the message here is that you're not limited to a single WOLA pipe. There is in fact considerable flexibility built in.)

The address space to address space structure means the two must be on the same LPAR. WOLA is *not* a cross-LPAR technology or a cross-platform technology.

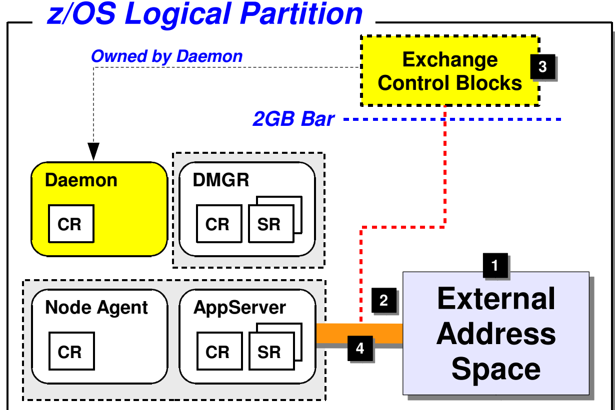
And because it's used for communication with a specific WAS server we have to be careful not to consider it as providing general reference to the WAS cell.

Now be careful ... once WOLA has passed off the data to a Java program in WAS, that Java program may then turn and route it to wherever it wants. So in that sense you can use it for the "whole cell." But the WOLA exchange itself is between the external address space and the WAS CR.

Ditto the external side ... once WOLA passes off to a CICS program it is free to use whatever CICS function it desires to pass off elsewhere.

Concept #3 - Registration

Before any external address space may use WOLA, it must first *register in:*



Registration ...

1. **Is *always* initiated by the external address space into the specific WAS server**
   There are different ways to accomplish this as we'll see
2. **The cell, node and server *short* name is given, along with a *registration name***
   The registration name is what identifies the WOLA transfer pipe. Since multiple registrations is permitted, this is how applications designate which pipe to use.
3. **The registration control block structure is built in the Daemon shared space above the 2GB bar**
   This is not in the Daemon address space. It is owned by the Daemon.
4. **Connection built and ready to use**
   Which brings up the inbound/outbound discussion

The key here is that the external address space has to request of WAS z/OS that it allow the access and build the connectivity infrastructure.

Inbound vs. Outbound ...

 © 2010 IBM Corporation

Concept #3 is about the very first thing that must be done before any WOLA exchanges can take place ... something called **registration**.

Registration is the act of letting the WOLA function know about the desire to establish a connection into the WAS Local Comm structure. This is what tells the Daemon to establish the control block structure up in the shared space. It's a very simple process.

Here's the key to understand: registration is *always* initiated by the external address space into the WAS z/OS environment. The process involves naming the specific server you want to connect to. That means the WAS z/OS environment -- the Daemon and the named server -- must be up and running when the registration in is attempted. If it's not, you'll get a return code and reason code that's very clearly documented in the InfoCenter as being a problem with the server not being up.

Registration can be done with the programming APIs, which we'll explore in a bit, or it can be done by supplied WOLA function that gets installed into CICS to support that environment.

You're not limited to a single registration. You may have multiple. The only restriction is they must be uniquely named. That unique name is used by the Daemon to keep the control block structures separate.

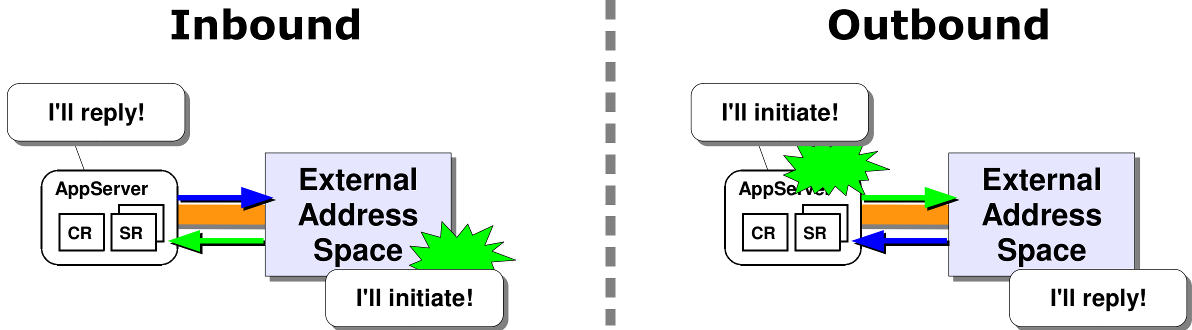
The name is also important because it's the way the programs on either side of the WOLA pipe reference which pipe to use. Since multiple registrations are possible, it's necessary to tell WOLA which of potentially many you wish to talk across. Again, this is merely a parameter on the programming interfaces.

© IBM Corporation 2010

Concept #4 - Inbound vs. Outbound

***Assuming registration is in place*, then this is all about which side *initiates* the conversation between programs:**

We think of this from the perspective of WAS z/OS:



This is important because many application considerations are a function of this

What APIs, what kind of Java program, security propagation ...

Java side ...

16 IBM Advanced Technical Skills © 2010 IBM Corporation

Key Concept #4 is one we wish you to really lock your mind on because when we get to the section on the native programming APIs this becomes a very helpful distinction. It's the concept of "inbound" vs. "outbound" calls.

The key here is who ***initiates the first call*** after registration is in place.

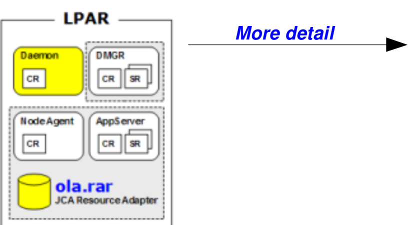
The words "inbound" and "outbound" are used with WAS z/OS as the point of reference:

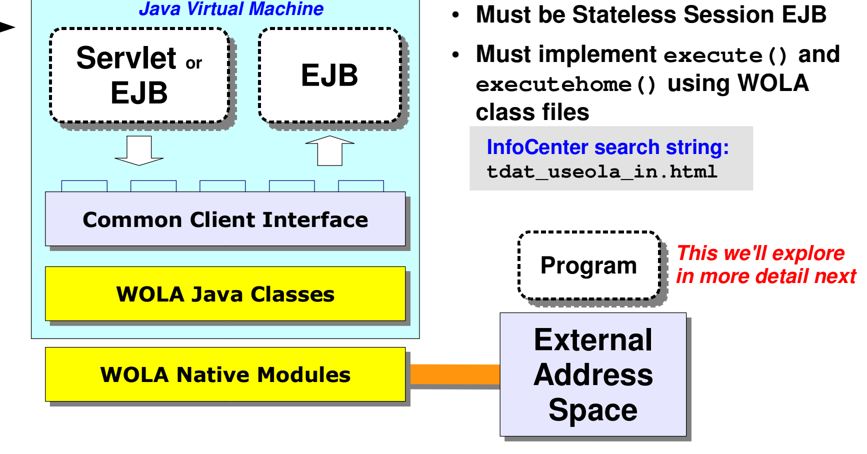
- **Inbound** -- the external address space initiates the first call into WAS z/OS after it performs the registration. The Java program in WAS z/OS responds.
- **Outbound** -- the Java program in WAS z/OS initiates the first call out to the external address space after the external has done the registration. The external program responds.

At this point you don't need to worry about why this is an important distinction. Just keep in mind the notion of this being from the perspective of WAS z/OS, and the key is who initiates after the registration is built.

Concept #5 - The Java Side of the Application Question

Much of WOLA is hidden by a standard JCA resource adapter. But a few things do come into play:





Inbound

- Must be Stateless Session EJB
- Must implement `execute()` and `executehome()` using WOLA class files

InfoCenter search string:
`tdat_useola_in.html`

Outbound

- Servlet or EJB
- Use standard CCI methods
- Pass in key parameters related to WOLA (i.e. register name, target program name)
- WOLA specifics are generally well hidden from the application

InfoCenter search string:
`tdat_useoutboundconnection.html`

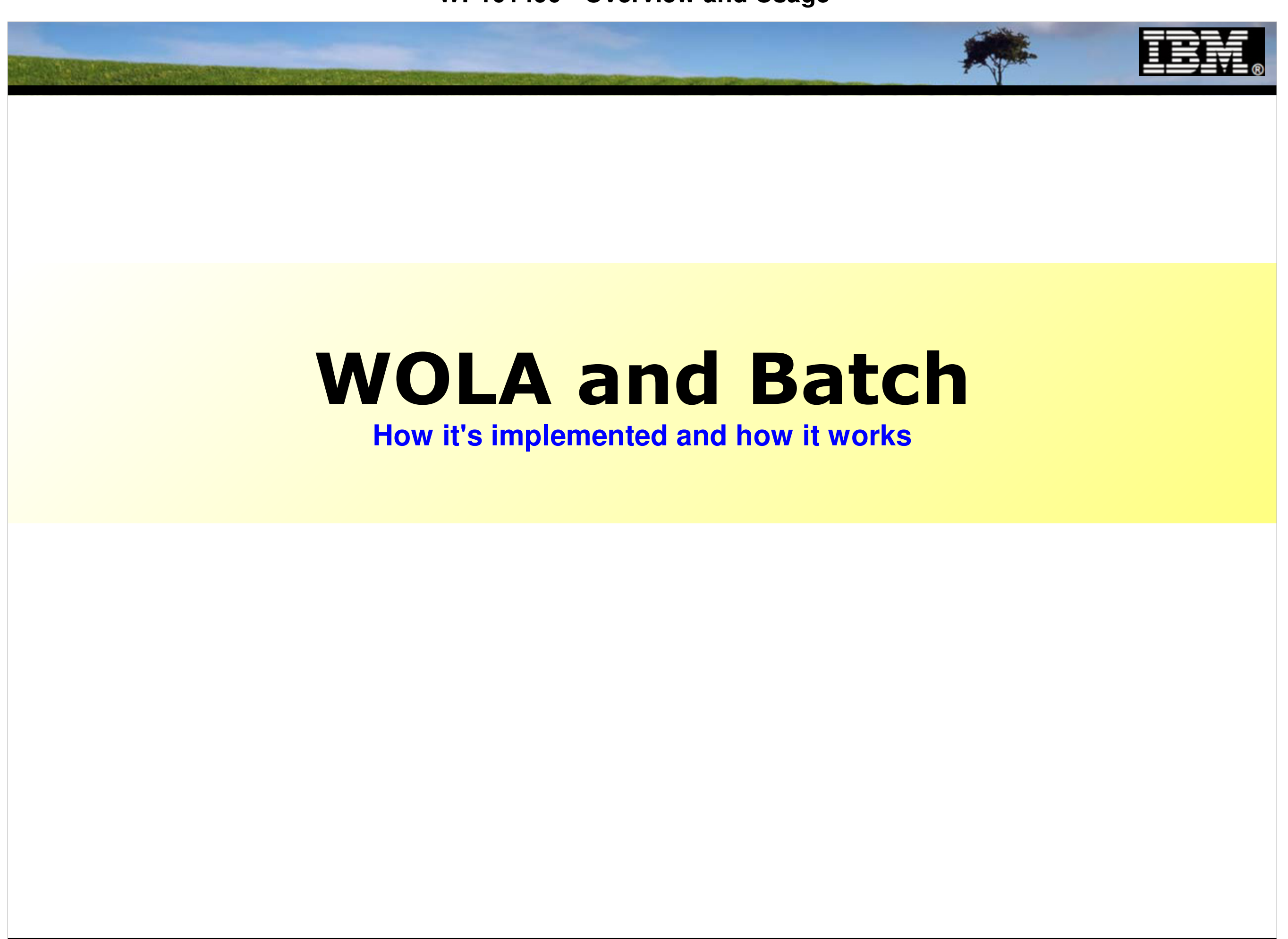
Not complete transparency, but a considerable degree of shielding Java from this very low level interchange

17 IBM Advanced Technical Skills © 2010 IBM Corporation

The fifth and final key concept has to do with the Java side programming considerations. We're not going to spend a lot of time on Java coding considerations because the focus of this document is non-Java. Suffice to say the InfoCenter has some good write-ups on this (see the "search strings" indicated on the chart).

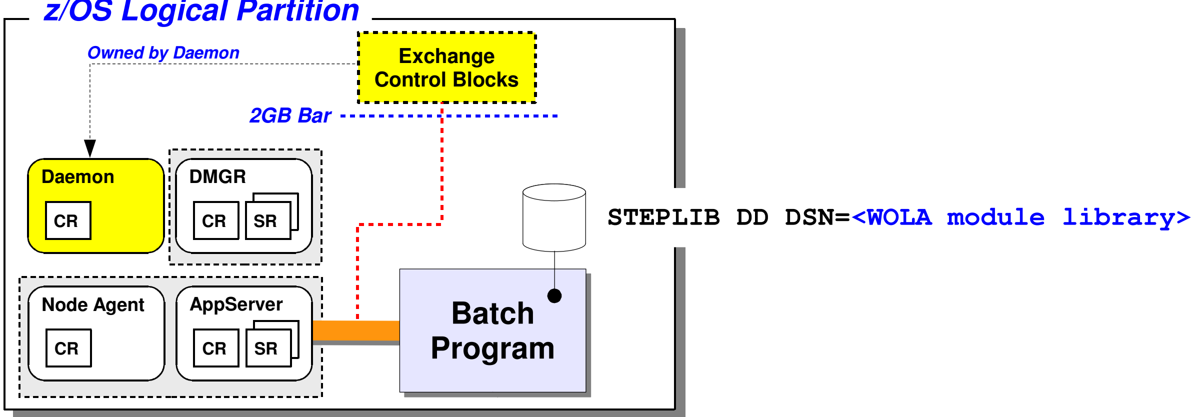
But the key message here is that the low-level specifics of WOLA are well hidden behind standard Java interfaces. For outbound calls it's the Common Client Interface (CCI), and for inbound calls it's the `execute()` and `executehome()` interfaces implemented with the supplied WOLA class files.

There's not complete transparency, just like there isn't for JDBC or JMS. There has to be some degree of application layer awareness of WOLA. But on the Java side it's limited to names to use in parameters. The Java program does not need to write to the APIs we'll be discussing next.


IBM
WOLA and Batch
How it's implemented and how it works
18
IBM Advanced Technical Skills
© 2010 IBM Corporation

Batch Program Access to WOLA Modules

This is easy ... copy the modules out to a module library and make it available to the batch program -- STEPLIB or LINKLIST



This merely gives your batch program *access* to the APIs

The next step is to actually *use* them ...

First step, registration ...

 © 2010 IBM Corporation

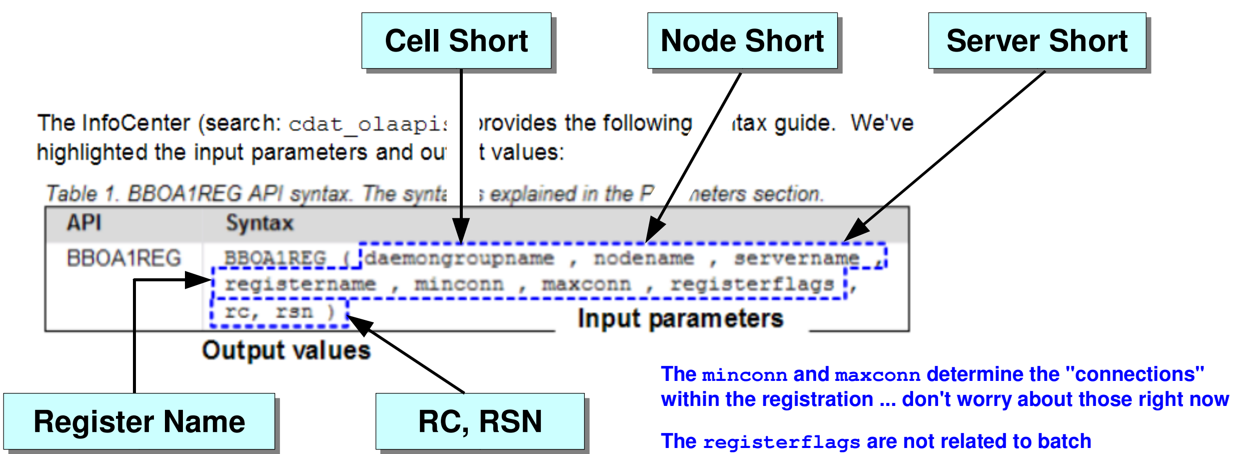
We start with batch access (rather than CICS or IMS) because it's the most basic and allows us to show WOLA at its most basic. CICS and IMS have various facilities that WOLA exploits to differing degrees. We'll explore all that when we get to the sections on CICS and IMS.

For now let's settle our minds on simple batch, and in so doing we can see the bare bones of WOLA usage.

© IBM Corporation 2010

Registering Into the WAS Server

The first step in any processing ... the external AS *always* registers *in*, never the other way around. From the "Primer" document at WP101490:



The `minconn` and `maxconn` determine the "connections" within the registration ... don't worry about those right now

The `registerflags` are not related to batch

Registration is one of the easier APIs to work with. It either works or it does not work, and if not the RC, RSN in the InfoCenter is really clear about the problem

The author of this presentation *loves* the InfoCenter's RC, RSN code write-ups ... perfectly clear and easy to understand

BBOA1INV ...

20 IBM Advanced Technical Skills © 2010 IBM Corporation

Recall that we said that registration is a key first step, and that it's always done by the external address space into WAS. We also said that registration was specific to a given server in WAS z/OS.

Well, no surprise, for a batch program to perform the registration it needs a programming interface (API), and that's one of the things supplied with WOLA -- a set of native APIs.

The supported languages are: COBOL, C/C++, PL/I and High Level Assembler.

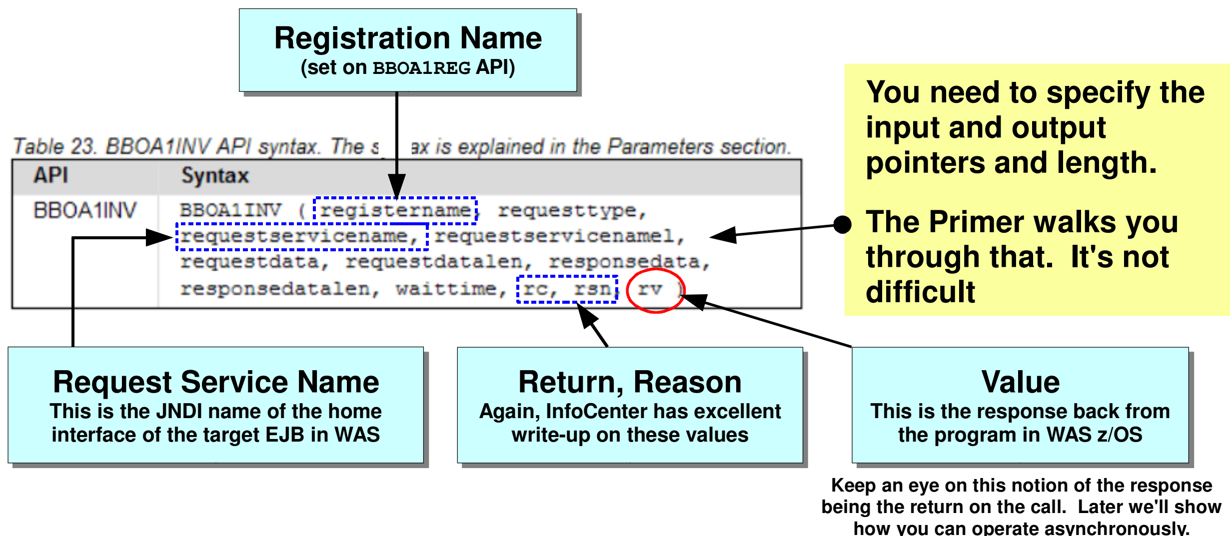
What you see on this chart is a bitmap clip from the InfoCenter (search: `cdat_olaapis`) and included in the WP101490 "Primer" document. It shows the API map -- command and parameters). The point of this chart is to show that the API for registration is not that complicated, and that the parameters make some sense with respect to what we've discussed already:

- BBOA1REG is the name of the API used to register. BBOA1URG is used to unregister as we'll see in a bit.
- The first three parameters that are passed in are cell, node and server *short* name. That's what's needed to isolate on precisely which address space you wish to connect to with the registration.
- The next is the registration name. As we mentioned before, that's important because the application layer needs to know which WOLA pipe to talk over. They would use this name.
- The next three -- minconn, maxconn and registerflags -- we'll defer until later.
- Finally rc and rsn are output values. This is what gets returned by WOLA. 0 is good, others may indicate an issue. The InfoCenter APIs page has a *wonderful* writeup on RC and RSN values and what they mean.

The "Primer" document walks you through a series of exercises on all the APIs, starting with registration.

Inbound: Invoking the Target EJB in the WAS Server

The BBOA1INV API is a "basic" API ... very simple to use ... it invokes the named EJB in the target WAS server:



This API can be looped over and over again within the same registration ...

Looping ...

21 IBM Advanced Technical Skills © 2010 IBM Corporation

Now we start to see where the "inbound" and "outbound" concepts come into play. It turns out the APIs tend to organize around inbound and outbound. This API -- BBOA1INV -- is an inbound API. It is used to "invoke" the target EJB in the server.

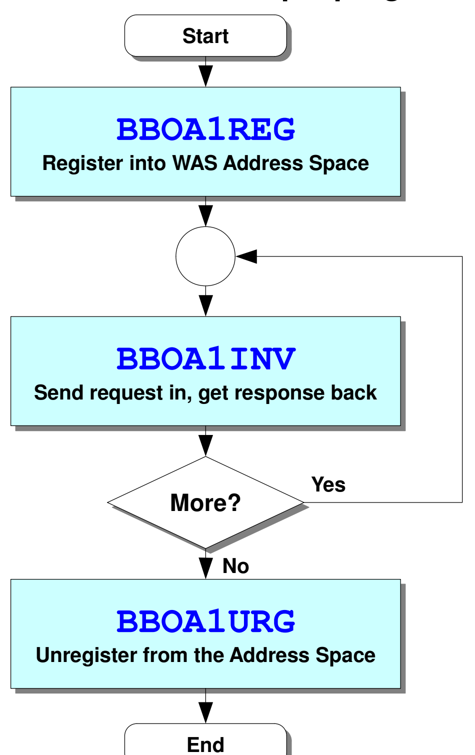
Let's quickly walk through the API:

- BBOA1INV is the name of the API used to register.
- The first parameter is the registration name. The batch program just performed that on the previous chart. That name can be maintained in a variable and used for this API. That's exactly what the Primer document shows.
- The second parameter (requesttype) is always "1" for EJB. The InfoCenter says that.
- The next parameter -- requestservicename -- names the target of the invocation into WAS z/OS. The value you put there is the JNDI name of the EJB you wish to invoke. Think about that for a moment. Within the Java EE environment objects are known by their JNDI names. Here -- from a batch program -- you are naming the target with a JNDI name. That means that somewhere the WOLA function turns the WOLA call into an IIOP call to invoke the target. That done by the supplied ola.rar resource adapter. Again, shielding the Java application from the details of WOLA.
- The next few parameters are used to indicate the storage location and length of the message. You can programmatically derive this values and plug them into the API use. The Primer shows this as well.
- Return, Reason and Value -- these are *output* values. The InfoCenter has a nice writeup for the RC and RSN. *The rv output value is the response back from the EJB.* Stop and think about that for a moment ... that means this API -- BBOA1INV -- must wait for the Java program to return the repsonse. It does. That means this API operates *synchronously*. But not all do ... later we'll see examples of "advanced" APIs that offer you the chance to operate *asynchronously* for even better performance.

© IBM Corporation 2010
Version Date: 11/12/12
WP101490 at ibm.com/support/techdocs

Inbound: BBOA1REG, Loop BBOA1INV, BBOA1URG

That's the most simple programming structure:



A registration may be used over and over again before tearing it down

Loop as many times as you need for the program requirements

The BBOA1INV is making some assumptions to keep things simple

- **Synchronous ... wait until response received**
- **Get connection, use, tear down connection**
- **(This is where "advanced" APIs come in)**

When done, tear down registration

Simple, easy, effective for inbound exchanges

API map ...

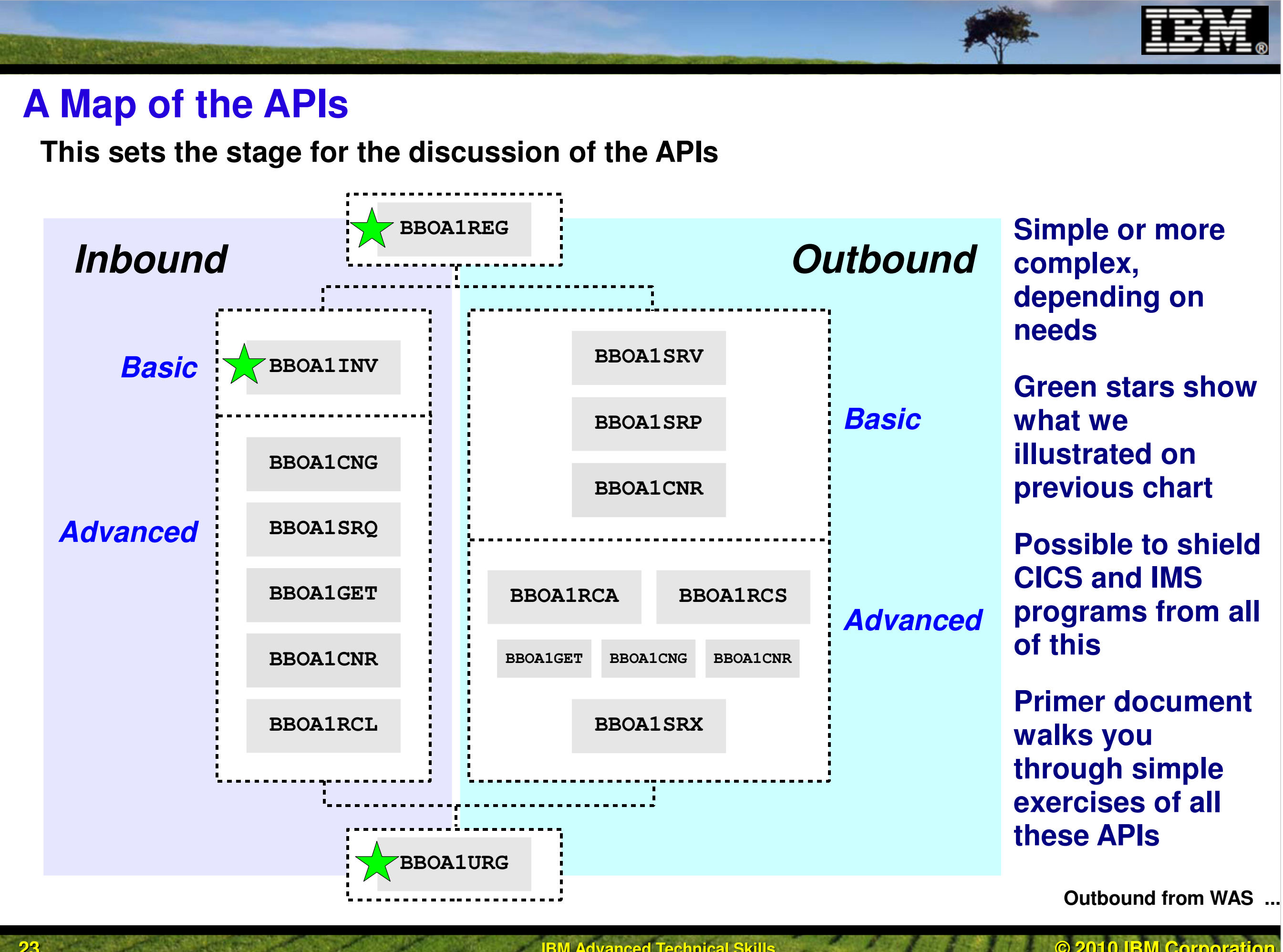
22 IBM Advanced Technical Skills © 2010 IBM Corporation

Batch processing is often about performing again and again the same action against different sets of data. And that's exactly what we see possible with WOLA -- your program can loop repeatedly.

Note that the registration does not have to be built each time. Build it once, use it many times, and when you're through with it, tear it down.

The BBOA1INV API is an example of a "basic" API ... one that is fairly simple to use. To accomplish that the developers of WOLA made some assumptions about the behavior of the API. One such behavior is something we just touched on in the previous chart ... this API is *synchronous*, which means it waits on the Java side to respond before returning program control to your batch application. This may be exactly what you want to do. It certainly makes things simple. But if the Java-side has longer or more variable delays, you may want to consider operating *asynchronously*, which means program control returns immediately.

"Ah!" You think. "That means my program would have to come back and get the response at some future point in time, right?" And you'd be right. The "advanced" APIs have the capability to do that. It requires more programming on your part, but it provides enhanced performance in return.

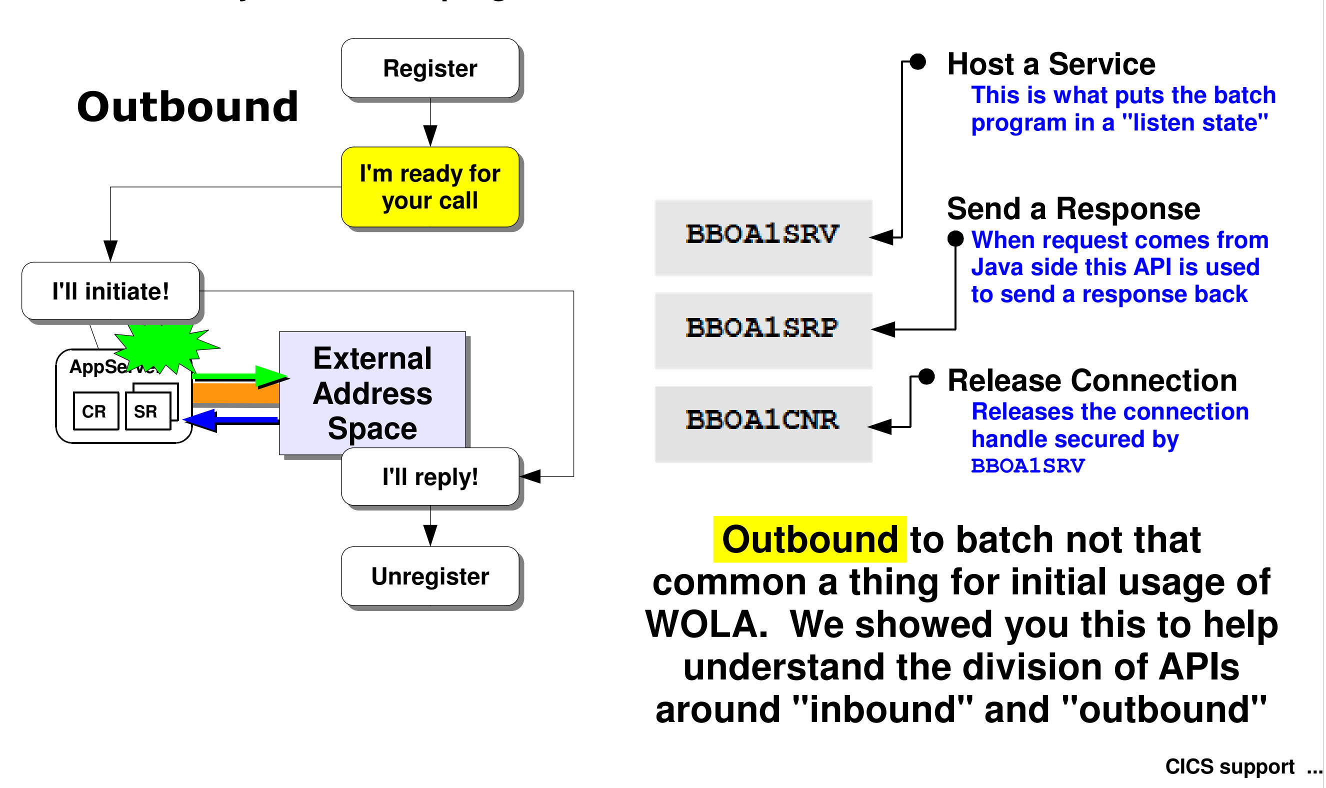


Here we see the complete list of APIs, organized around the "inbound," "outbound," "basic" and "advanced" concepts. The registration and unregistration APIs fall outside those classifications because, well, they are what they are they don't really apply to inbound or outbound, and their neither basic or advanced. They simply *are*.

We just looked at BBOA1INV. We saw that with that one API you can invoke the Java-side program over and over again.

So programming does not need to be complex. It can be very simple, as a matter of fact.

And we've yet to discuss CICS and IMS, which both have mechanisms to shield the programs from interaction with WOLA APIs. We'll see that in the upcoming sections.

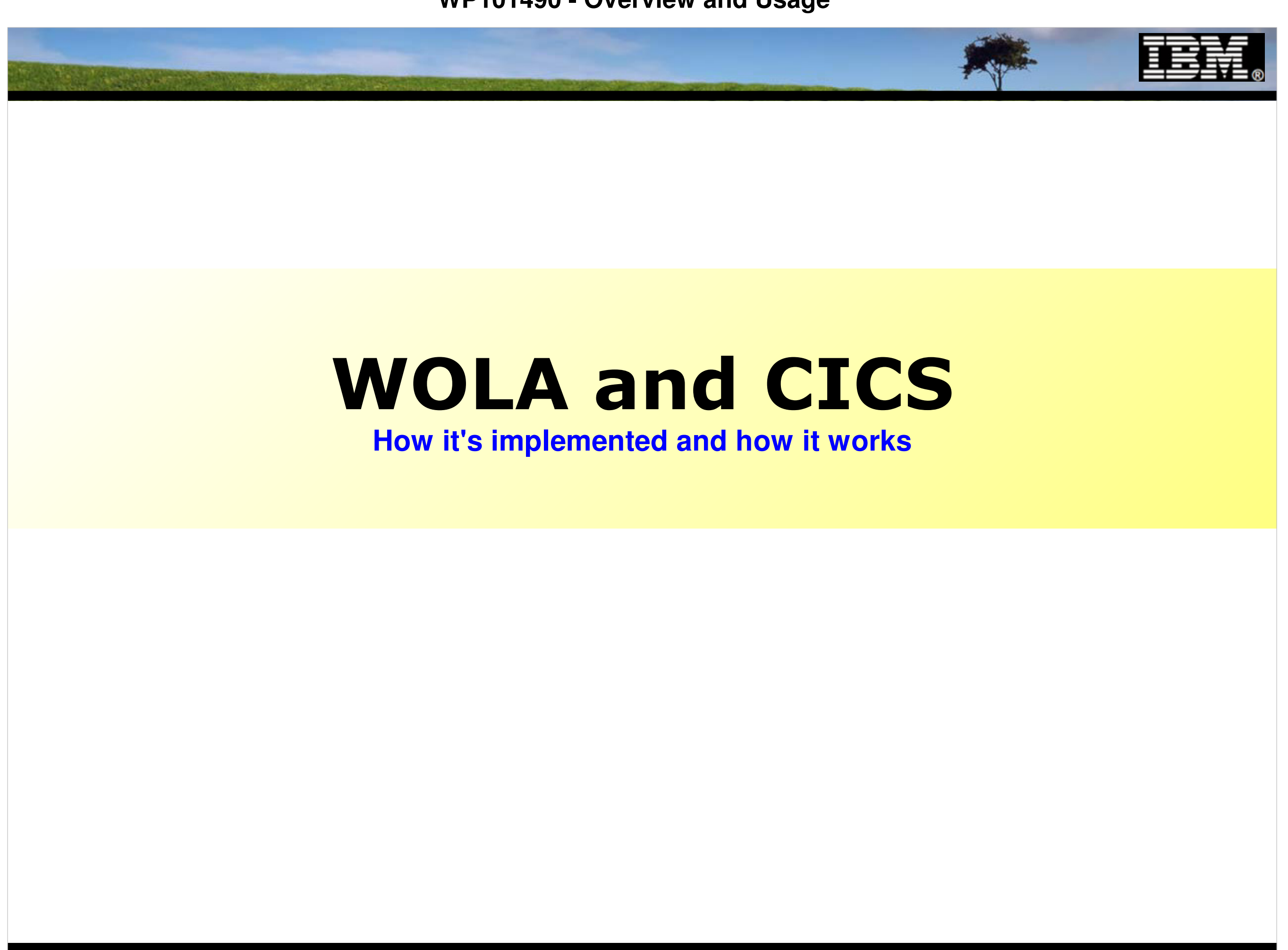


What about outbound calls from WAS z/OS to the batch program? The registration, as we noted, is always done by the external program. So that part is easy to understand. Must the batch program always be the one to initiate after registration?

There is a way to put the batch program into a "I'm listening and ready for your call, WAS, whenever you're ready to send it" state. That's done with the BBOA1SRV API, which "hosts a service." When WAS calls out to this hosted service, the batch program may then send a response with BBOA1SRP. The BBOA1CNR API is used to release the connection taken from the connection pool by the BBOA1SRV API.

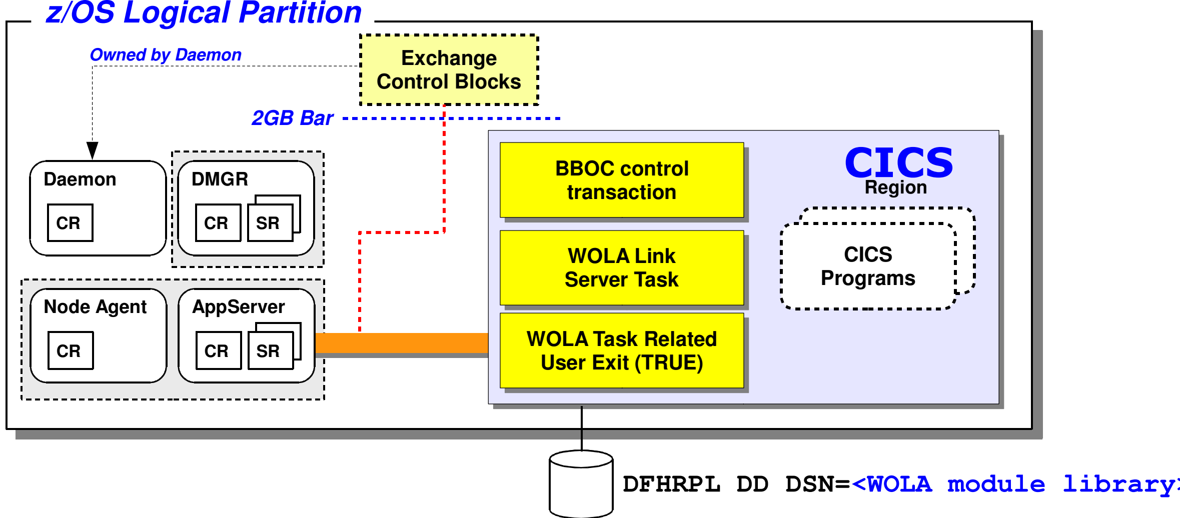
(Is there a way to re-use a connection over and over again? Yes. Again, advanced APIs.)

Outbound to batch is possible, but not likely to be that common. Outbound to CICS or IMS is another matter. So now we turn our attention to CICS, and see how things are implemented there.


IBM
WOLA and CICS
How it's implemented and how it works
25
IBM Advanced Technical Skills
© 2010 IBM Corporation

High Level Overview of CICS Support

The following picture illustrates the pieces:



TRUE -- the low-level heart of the WOLA support in CICS

Link Server Task -- a way of shielding CICS programs from the APIs and knowledge of WOLA

BBOC -- a useful command line transaction to start and stop various pieces of WOLA

Installation of each is standard CICS system programmer stuff ... very easy

Link Server Task ...

 © 2010 IBM Corporation

This picture provides a very high level view of the components that make up the WOLA support in CICS. We'll drill into more detail over the next few pages.

The WAS-side components are the same regardless of the external address space. Batch or CICS or IMS or ALCS ... it's always the same setup requirements inside of WAS z/OS.

There's three pieces to the puzzle here:

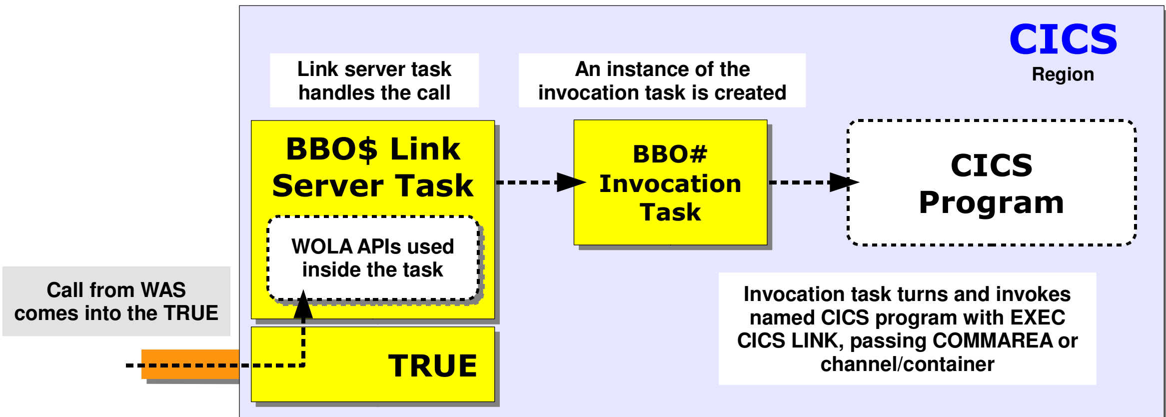
- The Task Related User Exit (TRUE) -- this is what provides the low-level module support for access to the Local Comm function of WAS z/OS. It is required in all cases. CICS can't use WOLA without it. And this has to be installed in any CICS region that wishes to use WOLA.
- The Link Server Task is what can be used to shield the CICS programs from the specifics of WOLA. It serves as a kind of "WOLA handler" which then turns and uses EXEC CICS LINK (DPL) of the named target CICS program. The Link Server Task is for outbound calls. Inbound calls, as well see, requires some coding to the APIs.
- The BBOC control transaction is something provided to make managing the environment easier.

This is all very straight-forward installation work. So again, setting it up is not the issue. Using it is the objective.

© IBM Corporation 2010

Link Server Task

The piece that allows you to shield CICS programs from knowldge of WOLA. It applies to outbound calls ...



The target CICS program has no awareness of WOLA. It sees that another program invoked it using standard CICS routines. The BBO$ / BBO# activity is automatic.

The program's response is returned to WAS automatically as well.

There are details and optional changes to behavior we're not showing you here. The key concept is that there is a way to shield CICS programs and the Link Server Task provides it for *outbound* calls.

The BBOC transaction ...

The developers of WOLA understood that any solution that required updating and recompiling the program code would face resistance in the market. Further, they understood that many CICS programs are written to be invoked using the CICS Direct Program Link (DPL) facility. There is a long-established means of passing data (COMMAREA). So the developers decided to provide a shielding function called the Link Server Task that would drive CICS programs with a DPL, leaving the invoked CICS program no more the wiser that WOLA was involved.

And that's what the Link Server Task does -- it accepts outbound calls from WAS into CICS, then turns and issues EXEC CICS LINK against the named CICS program, passing either COMMAREA or Channel/Container.

One other piece of this puzzle is the Invocation Task ... this is spawned to actually do the EXEC CICS LINK. The reason why this is separate from the BBO$ link server task is if specific security identities are required on the DPL to the target CICS program, then a separate invocation task for each is used. If your security requirements indicate invoking with the same ID then you can use specify the re-use of the invocation task.

There's always a tradeoff on these sorts of things -- flexibility and easy of use vs. efficiency and speed.

The Link Server Task also plays a role in the new 7.0.0.12 support for two phase commit processing outbound from WAS to CICS. So if 2PC out to CICS is needed, then the link server task is indicated.

© IBM Corporation 2010

The 3270 BBOC Transaction

This provides the tools to do many things with the WOLA support in CICS:
InfoCenter search string: `rdat_cics`

TRUE Related

```
BBOC START_TRUE <parms>
BBOC STOP_TRUE <parms>
```

You may automate this by including the TRUE in the Program List Table Post-Initialization (PLTPI)

Link Server Related

```
BBOC START_SRVR <parms>
BBOC STOP_SRVR <parms>
```

Parameters include ability to process registration on link server task start, set security and transaction flags and other options

If you don't use the link server task you do not need to start the link server

Registration Related

```
BBOC REGISTER <parms>
BBOC UNREGISTER <parms>
```

Provides a way to process a registration into WAS z/OS outside the program itself.

Or use `BBOA1REG` in the program

Or use the link server task which can also process registration

Point here is not the details of this, but rather what BBOC offers

CICS inbound ...

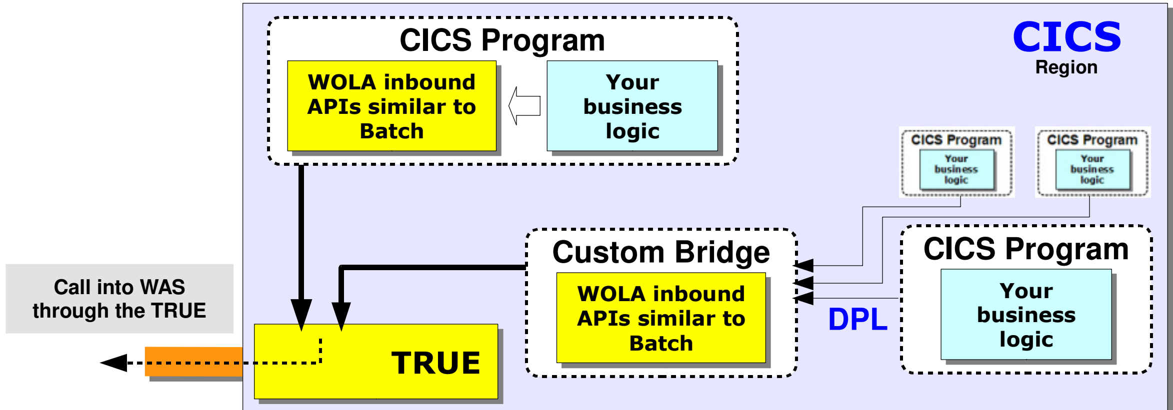
 © 2010 IBM Corporation

The 3270 BBOC control transaction is supplied with the WOLA CICS support and does a few handy things for you. All of these things can be done using other mechanisms, so the BBOC transaction isn't strictly required to operate. But it is handy. The chart spells out the uses of the transaction.

© IBM Corporation 2010

When the CICS Program Initiates an Inbound Call

The Link Server Task does not come into play. Therefore, at least some coding to the WOLA APIs is necessary. But here *you* can provide some shielding ...



Embedded API Usage

- API's in the compiled program
- Normal WOLA Inbound APIs
- Can offload register/unregister to manual BBOC if you wish

Custom Bridge Program

- Modular design ... isolate WOLA from business logic
- Multiple programs could use DPL to drive WOLA
- Bridge code then uses APIs

Transaction and Security ...

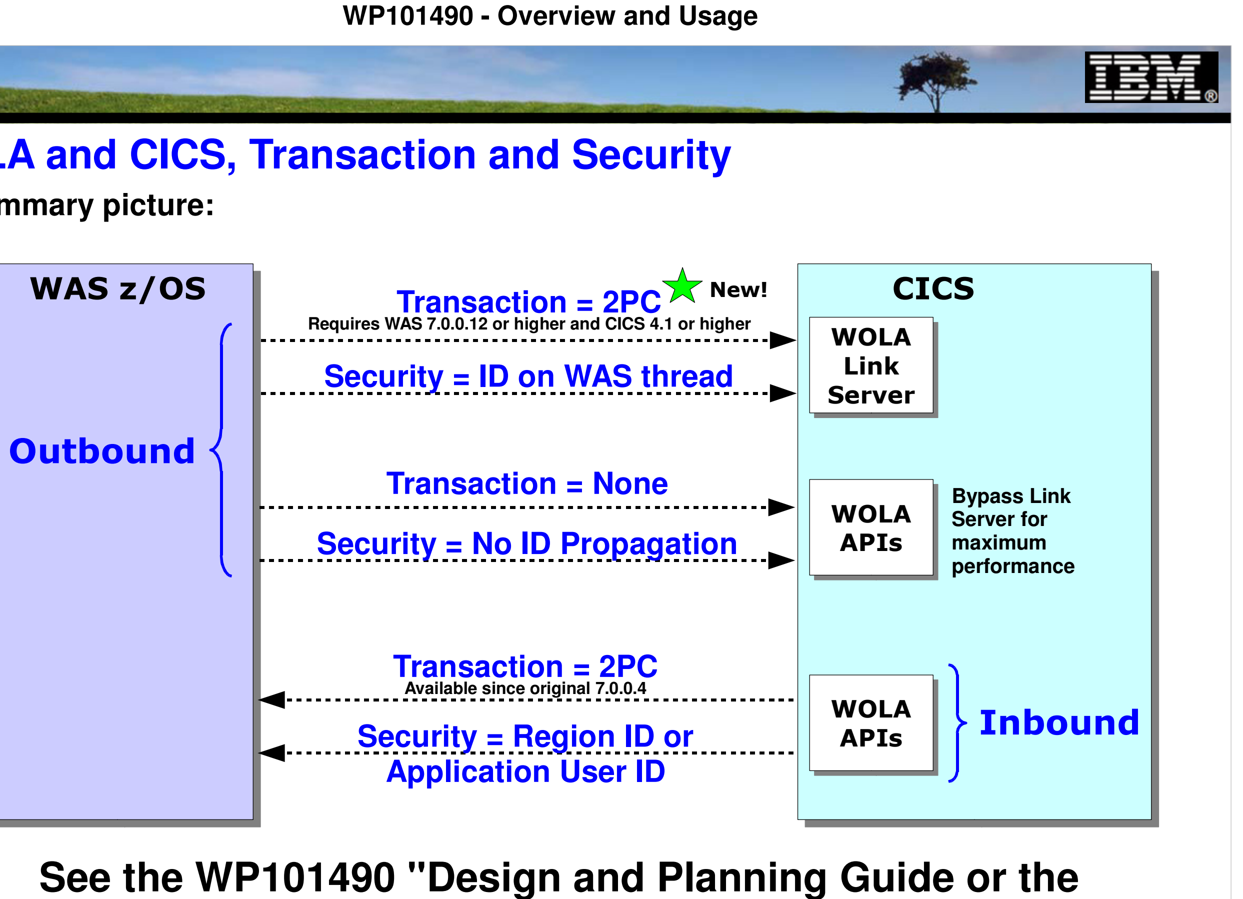
Let's turn our attention to inbound calls. Like batch, this is going to involve some degree of coding to the APIs. The Link Server Task does not asist in calls inbound to WAS. The TRUE is still needed (always needed), but the Link Server Task does not play a role.

Note: again, responses to calls outbound from WAS where the link server task is involved is a different matter. The link server task in that case does process the response and pass it back to WAS. What we're talking about here is, again, the *initiation* of the exchange.

Some coding to the APIs is needed. But where in the architecture will that reside?

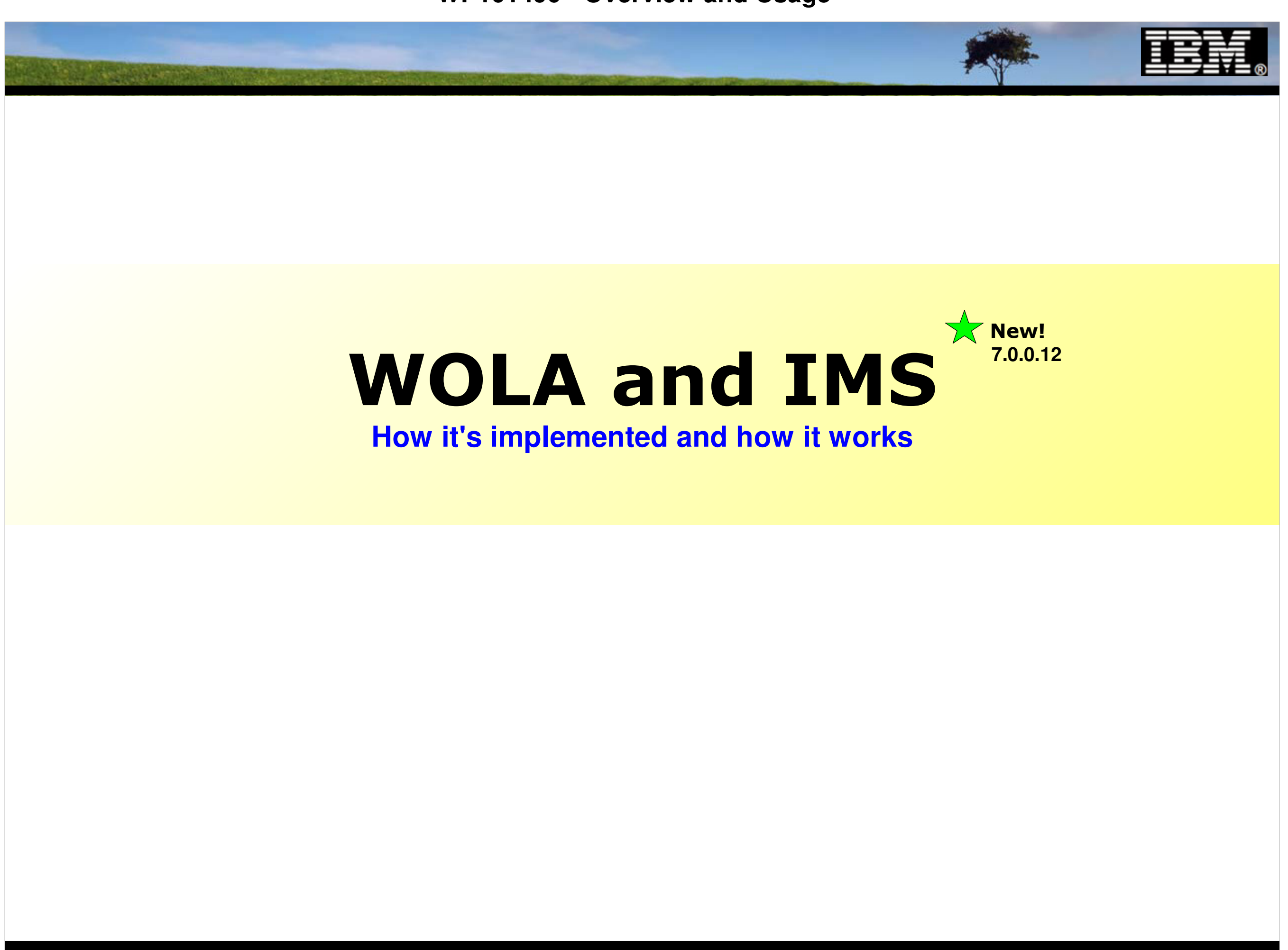
There are two broad approaches:

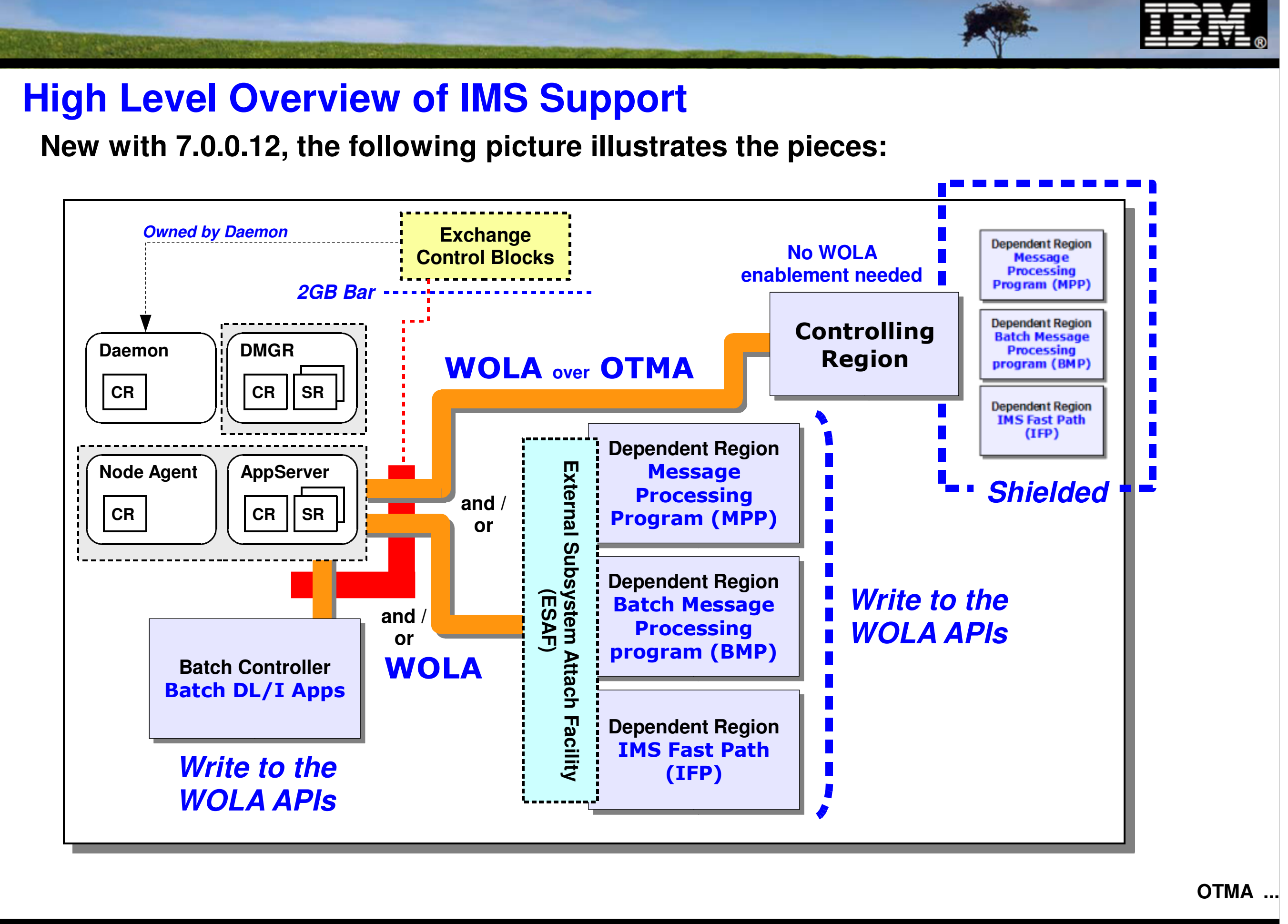
- Embed the WOLA API processing in the CICS program itself. This is easy enough to do if you're writing a new application, or can easily modify an existing one. But it's a bit more problematic for existing programs where the mandate is to leave it untouched for modifications.
- The second option is to write a kind of custom WOLA bridge program this is DPL-able by your programs. In this sense it becomes a kind of "WOLA Service" to existing CICS programs to utilize as your needs require.



This chart is summarizing the transaction and security support offered with WOLA and CICS. The InfoCenter has significant detail around all the information provided on this chart. But in summary:

- For outbound where the Link Server Task is used, assuming you're at the 7.0.0.12 level or higher and CICS 4.1 or higher, then you can have CICS participate in a WAS-initiated global transaction with two phase commit. Prior to 7.0.0.12 the support from WAS into CICS was limited to SyncOnReturn. That's been addressed with WAS 7.0.0.12 and CICS 4.1.
- For outbound where the Link Server Task is used, you can assert the identity of the WAS thread of execution into CICS and have that ID be used for the EXEC CICS LINK to the named CICS program. For maximum performance you can forego asserting the WAS thread ID into CICS and have the ID used to start the Link Server task used each time.
- For outbound without the Link Server Task there is no assertion of identity nor any assertion of transactional context. This is just like it is with batch.
- For inbound to WAS you can have WAS participate in a CICS global transaction with full two phase commit support.
- Inbound may also assert the CICS region ID or the application user ID.


IBM
New!
7.0.0.12
WOLA and IMS
How it's implemented and how it works
31
IBM Advanced Technical Skills
© 2010 IBM Corporation



This is a somewhat busy chart, and we'll break this down into the simpler components in the following charts. What this chart is trying to do is capture on a single page the various options for WOLA support with IMS.

Like CICS, the IMS support provides a way to shield the IMS programs from having to code to or otherwise know about the WOLA APIs. That's the "WOLA over OTMA" section. OTMA is an IMS architectural component that allows access to the IMS controlling region over the Sysplex XCF facility. Once over the OTMA interface line it's all IMS from there. No changes to anything needed.

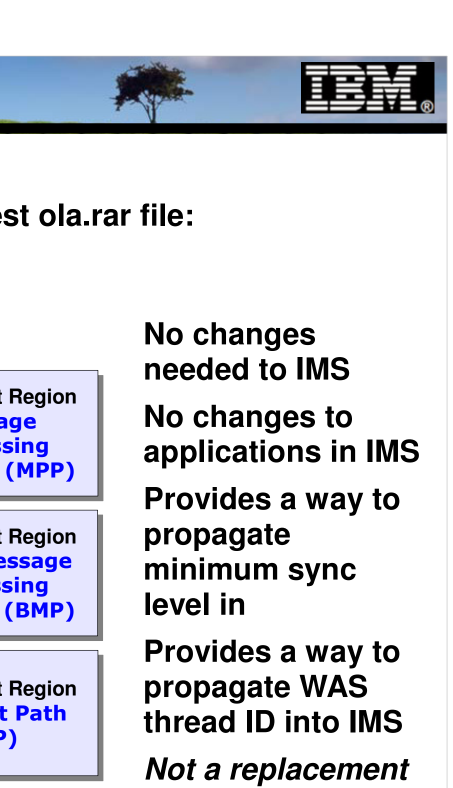
It is possible to use WOLA to talk directly to programs in the dependent regions. For inbound to WAS this is viewed by IMS as an ESAF call, and the WOLA APIs have been modified to do this automatically. You still have to write to the APIs to perform the call inbound to WAS. But you don't have to worry about the ESAF elements of this as it's part of the WOLA IMS support structure.

Outbound from WAS to IMS dependent region programs involves coding to the APIs. We'll touch on that in a few charts.

Finally, you can use WOLA to talk to Batch DL/I applications. This is essentially just like other batch so we won't dwell on this much.

The OTMA Support

To use the OTMA support it's important to be using the latest ola.rar file:



- No changes needed to IMS
- No changes to applications in IMS
- Provides a way to propagate minimum sync level in
- Provides a way to propagate WAS thread ID into IMS
- ***Not a replacement for IMS Connect***
  It's complementary

`setOTMAServerName()`	XCF Server name for IMS control region
`setOTMAGroupID()`	XCF Group name for IMS control region
`setOTMASyncLevel()`	To set SyncOnReturn or SyncLevel1 (CM0 or CM1)

Inbound ESAF ...

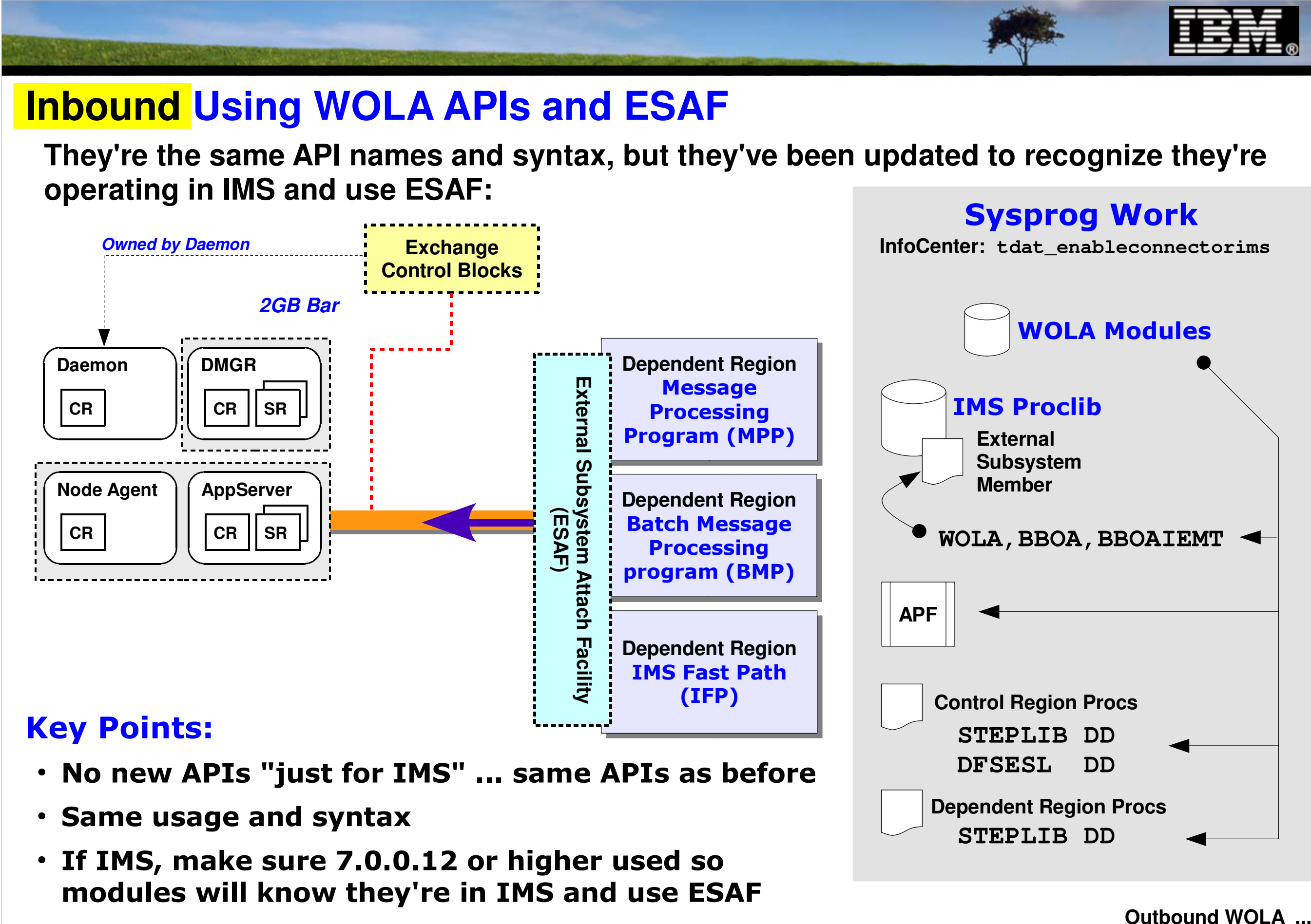
33 IBM Advanced Technical Skills © 2010 IBM Corporation

OTMA is a call interface on the IMS controlling region and is accessible by other address spaces. It provides a programming interface to programs and transactions running inside IMS. Access to the OTMA call interface is across cross coupling facility (XCF).

The WOLA developers implemented this by extending the WOLA JCA resource adapter and lower-level WOLA code to access XCF and be able to address itself to the OTMA CI. No changes are needed in IMS to make use of this facility. That's what provides the shielding of applications from WOLA by the use of this feature.

It is important to use the RAR file delivered with 7.0.0.12 rather than an earlier copy. The 7.0.0.12 RAR file provides the methods to access the OTMA interface and specify the sync level desired. Once into the IMS CR it's all IMS from there.

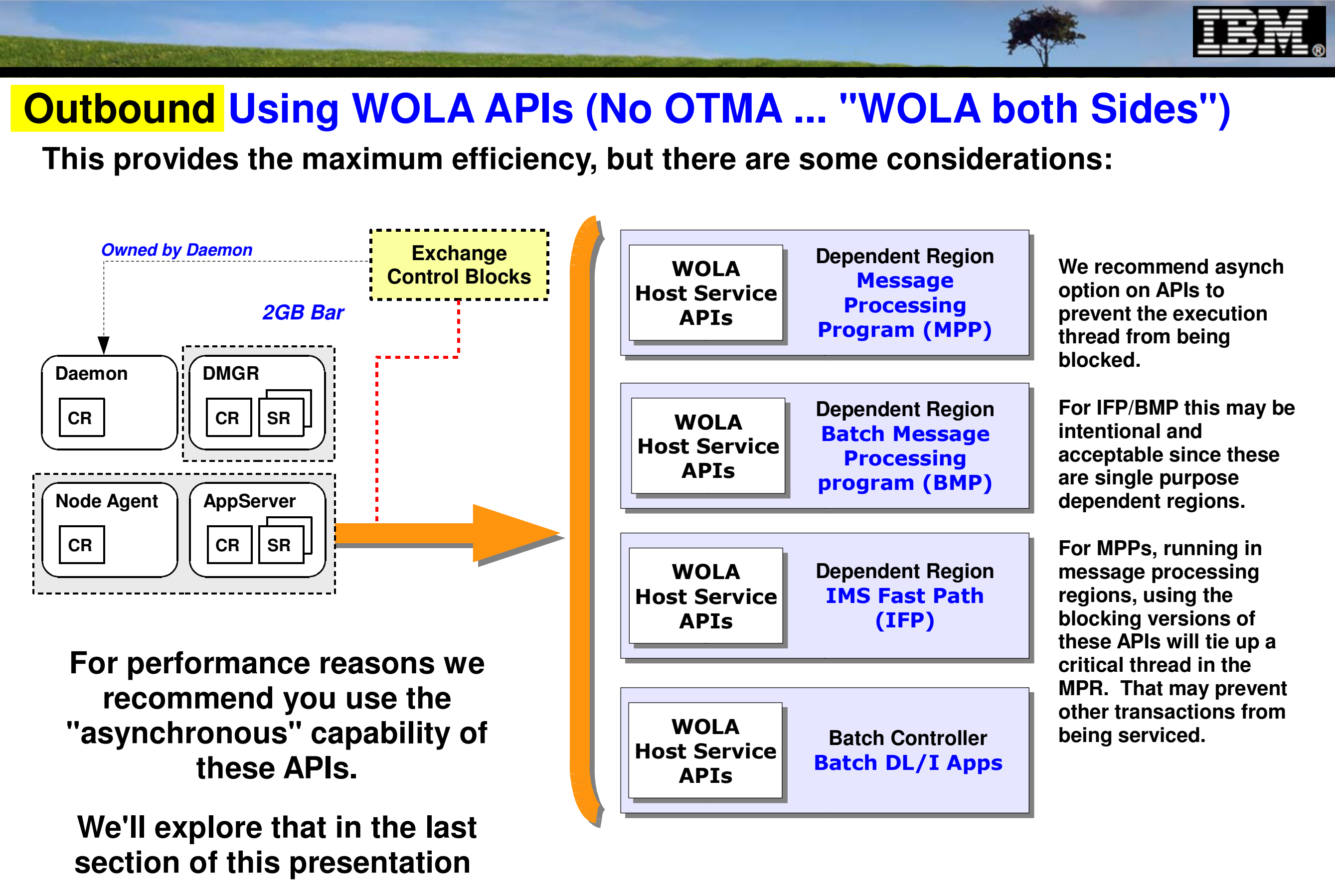
This provides a way to assert the WAS thread identity into the IMS region over WOLA. That's one of the differential functions of this over IMS Connect. We would caution that no reader conclude this is a replacement for IMS Connect. IMS Connect offers many other functional components above and beyond simple access. They are complementary technologies, not competing.



If you wish to access an application in WAS by going *inbound* with a call from IMS you would write to the WOLA APIs. These APIs, in turn, use the IMS External Subsystem Attach Facility (ESAF) function to communicate to the external subsystem. That "external subsystem" is WOLA. The system programmer work we show on the chart gives a hint to the relatively simple setup that's required to let IMS know about the WOLA external subsystem so the APIs work and talk to WAS over WOLA.

The key here is that there's no new "IMS only" APIs. The exact same APIs we showed you earlier apply to IMS as well. **But** ... it is important to make sure you use the 7.0.0.12 level or higher of the WOLA native modules when you do that sysprog work. That's the level that has the updated API modules so they recognize they're operating in IMS and use ESAF. Earlier levels of the modules will be simply confused. ☺

No new APIs to learn, no new concepts. WOLA is WOLA when it comes to the API syntax and usage.



We saw that outbound WAS to IMS can be done using the OTMA call interface. That works, but it won't be as fast as having the programs in the IMS dependent regions "host a service" using the WOLA APIs.

But ... do be careful. We recommend you investigate and learn the use of the asynchronous options on those APIs. The "basic" APIs assume synchronous control, which means the hosting IMS program thread has to wait between calls from WAS. That may be what you want, but it may not.

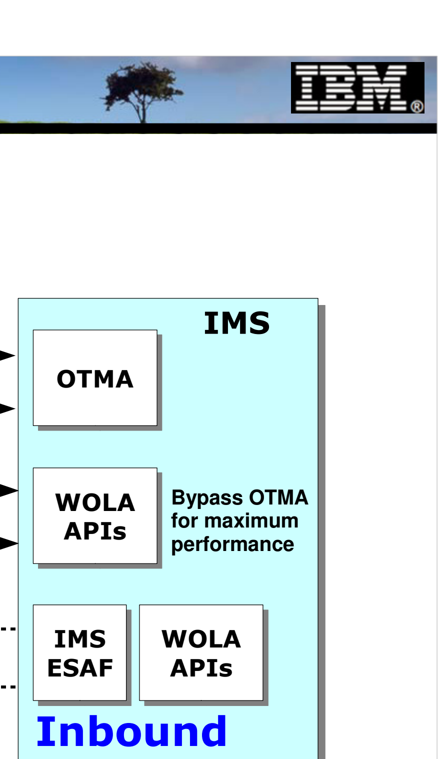
In particular, be careful of Message Processing Program regions and synchronous calls. There's a possibility of that resulting in the exhaustion of execution threads and no further transaction requests coming in.

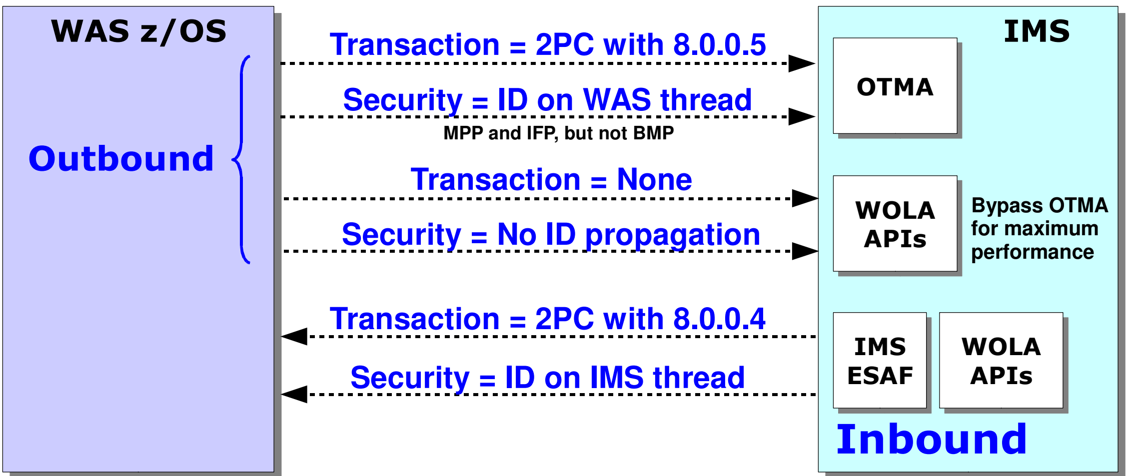
The asynchronous API options are not difficult, but they do impose a bit more programming effort on you for the IMS side. Squeezing high efficiency out of IMS applications has been the standard practice for a long time now, so the concepts should not be foreign at all.

We cover asynchronous API structures in the latter part of this presentation. And the Primer document on WP101490 does as well.

WOLA and IMS, Transaction and Security

A summary picture:





The transactional propagation issue is on the list of enhancements for the future.

See InfoCenter for details about security assertion requirements

Setup very similar to those required for traditional batch

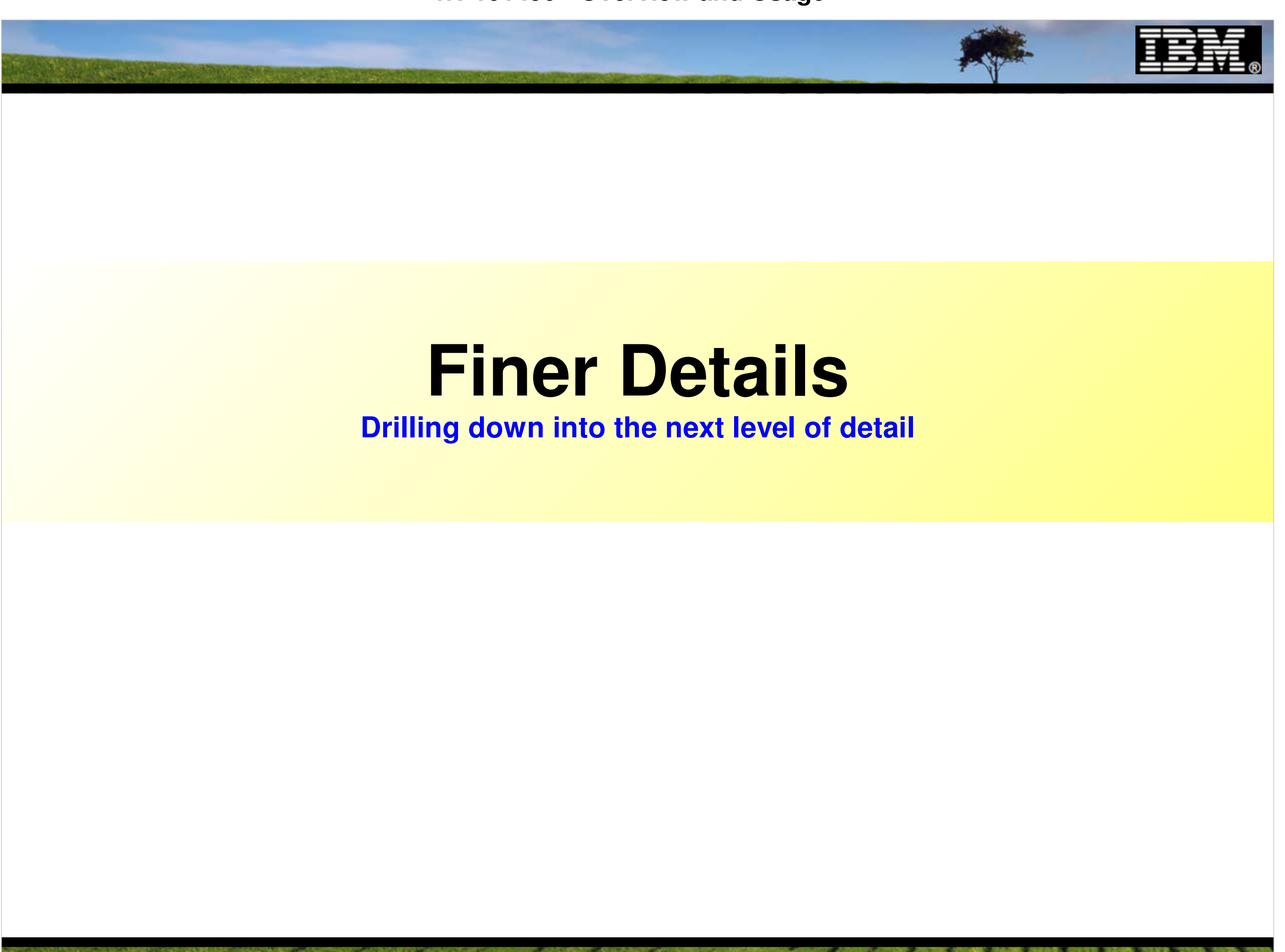
Advanced considerations ...

36 IBM Advanced Technical Skills © 2010 IBM Corporation

This is a chart similar to the one we offered for CICS, showing a summary of the security and transaction support offered for various combinations of things. Again, the InfoCenter has good detail behind all this.

For outbound to IMS over OTMA, the transaction level is SyncLevel None or Confirm, and the security context may be the ID on the WAS thread of execution. As the green bar on the chart shows, the transactional support offered is on the list of known requirements.

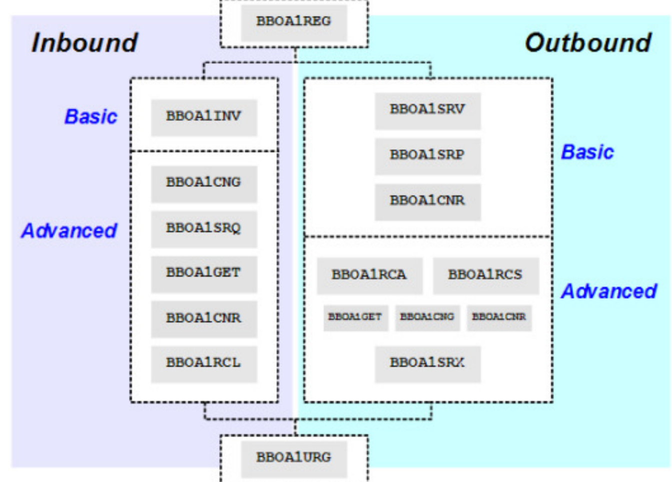
2PC when IMS asserts transaction into WAS from IMS Dependent Region applications was addressed with WAS z/OS 8.0.04. 2PC WAS into IMS over OTMA was made available with 8.0.0.5.



© IBM Corporation 2010

Reminder of the API Map Shown Earlier

We'll now discuss the Inbound/Outbound and Basic/Primitive issue a bit more ...



Inbound / Outbound

- Key is who initiates exchange
- Various things available to shield external from outbound APIs
  - CICS link server task
  - IMS OTMA
- Those shielding mechanisms imply *some* overhead; using the APIs can reduce that

Basic / Advanced

- One set is simple to use but it makes assumptions to keep things simple
- The other set is a bit more complex but it gives you far greater control, which can translate to greater performance

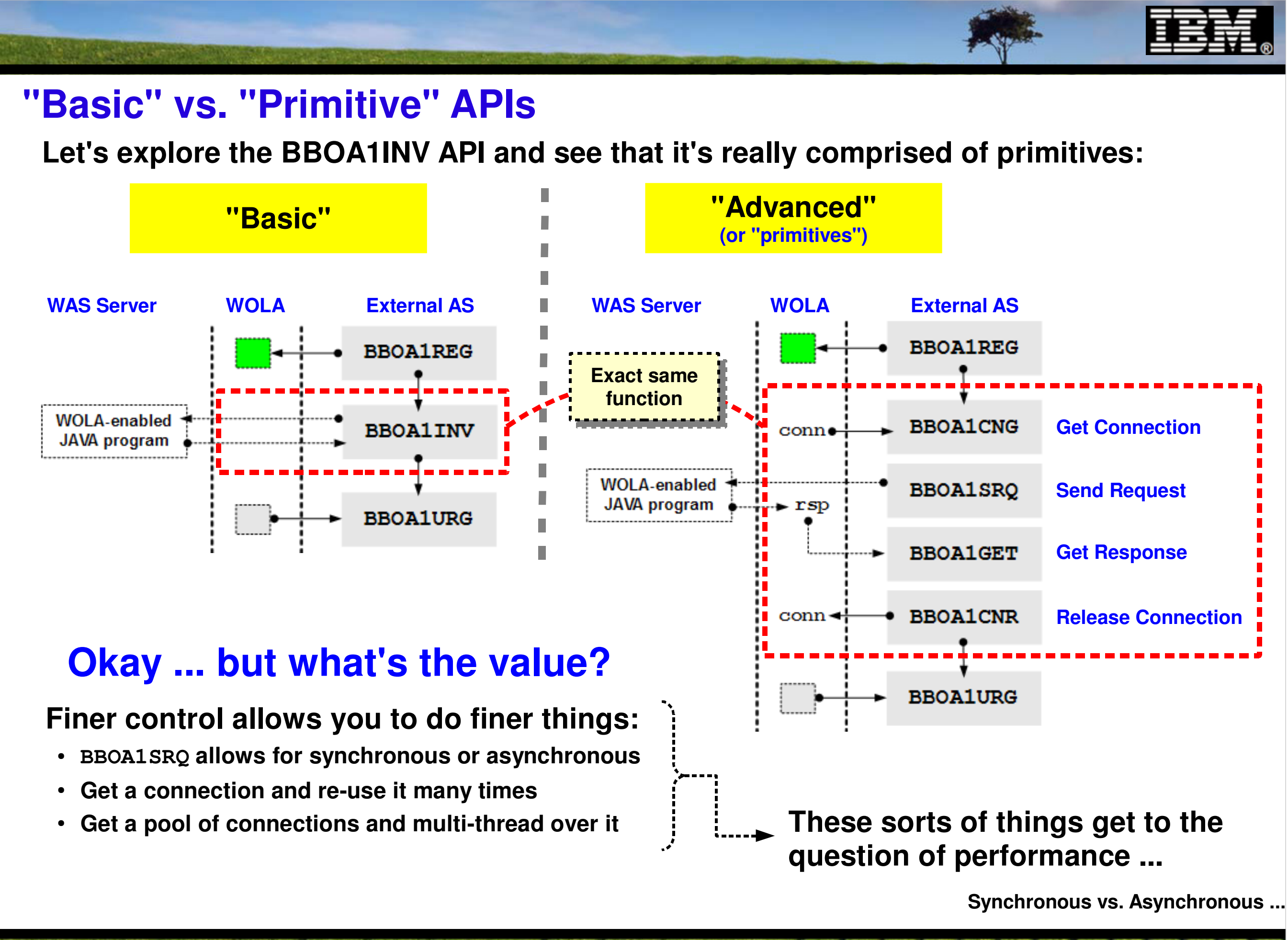
Basic vs. Primitive ...

38 IBM Advanced Technical Skills © 2010 IBM Corporation

This chart is just to remind you of the API map we showed earlier. In that map we broke down the APIs into four categories -- inbound vs. outbound, and basic vs. primitive.

We spoke at length about the nature of "inbound" and "outbound" and how that plays heavily into which APIs are used. And we saw that for CICS and IMS there are means of shielding programs from the outbound calls.

Those shielding mechanisms do imply a level of overhead. So there's a set of APIs to "host a service". But even those make certain assumptions that migh affect throughput. So WOLA offers "advanced" (or "primitive") APIs, which give you maximum control to tweak every bit of performance you can out of the exchange mechanism.



We'll use the BBOA1INV API to explain something about how these APIs work. We'll use the term "primitives" here rather than advanced because the InfoCenter tends to use "primitives" more frequently. In a sense it's a better term because it describes more accurately the relationship between them and the easier-to-use "basic" APIs.

You see, the "basic" APIs are really comprised of primitive APIs under the covers. Or, another way to put it is the basic APIs are simpler wrappers around the primitives. But to do this it's necessary to make some assumptions about the behavior of the basic That's how you make it simpler. By providing access to the primitives that comprise the basic you the user has access to the finer-grained control you may desire.

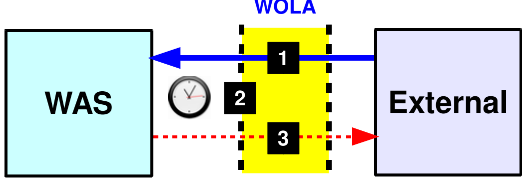
Take as an example the BBOA1INV API. The assumptions it makes is that when invoked the program will wait until WAS responds. That's what makes BBOA1INV *synchronous*. Further, it assumes that the connection from the connection pool used for the invocation is returned to the pool. That's simpler but there's a bit of processing underutilized by doing that. Better to hold the connection and use it over and over again.

Asynchronous operations and direct connection handling are attributes of the primitive APIs. And they make up the basic APIs.

Synchronous vs. Asynchronous

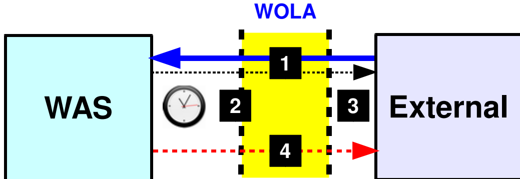
The APIs allow both. In general, synchronous is simpler. But asynchronous allows for potentially greater throughput:

Synchronous



1. External program calls WAS program
2. WAS program processes request. Program control is held from external processing thread until request returns.
3. WAS program responds

Asynchronous



1. External program calls WAS program. Program control is returned to external thread immediately.
2. WAS program processes request.
3. External program free to do other work or employ other WOLA connections (more on connections next chart)
4. WAS program responds at some future point.

The "basic" APIs operate synchronously. It's a simpler model.

The "advanced" APIs (sometimes called "primitives") are finer-grained subsets of the basics which allow asynchronous activity. ***But that implies your program goes back at some point and checks to see if a response has been received.***

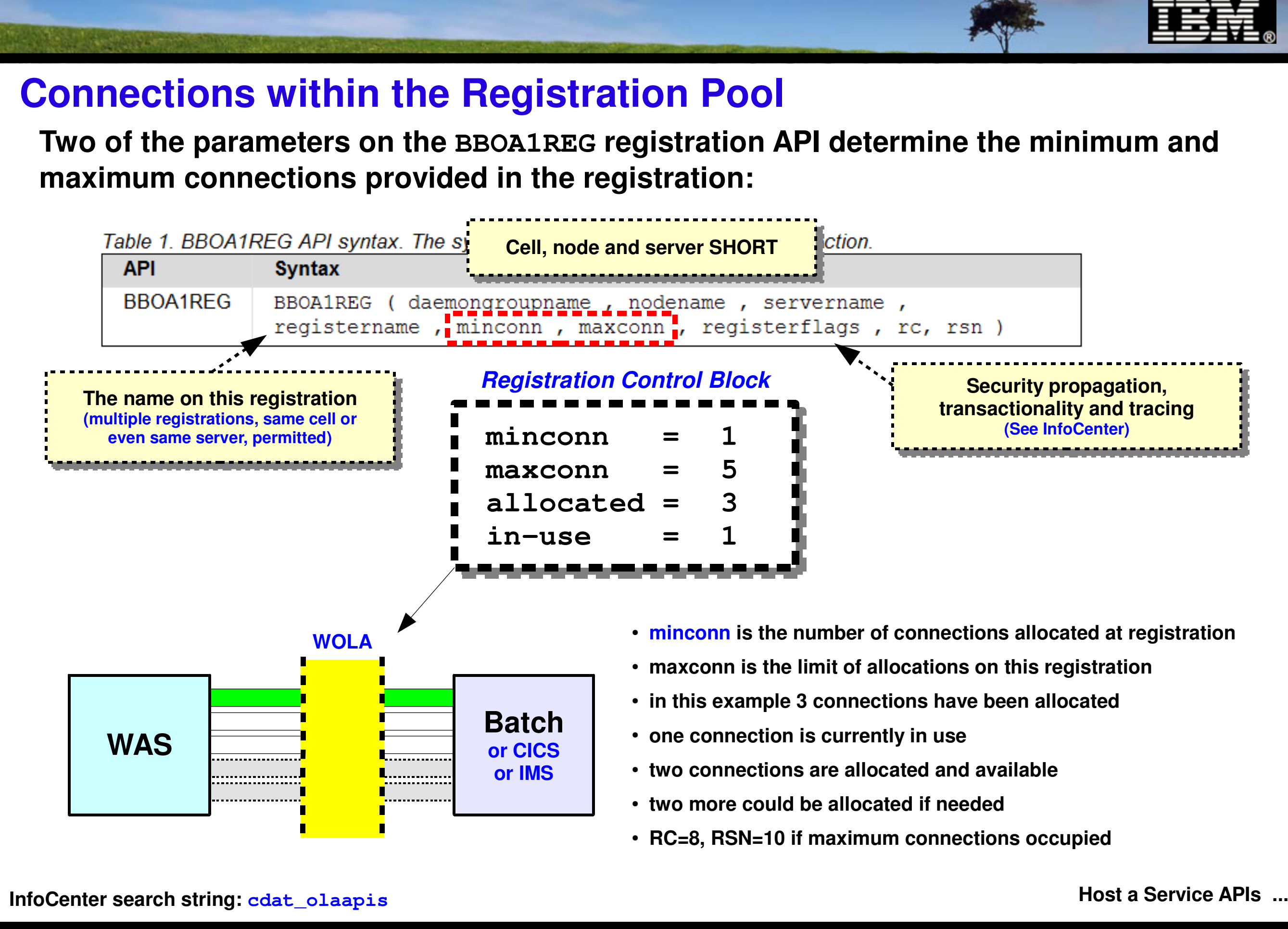
Connections ...

 © 2010 IBM Corporation

We've touched on the concept of synchronous vs. asynchronous earlier. He we illustrate it. It's all about whether the program control is returned to the program immediately (asynchronous) or held under a response is received (synchronous). Synchronous processing is easier; asynchronous has the potential for better performance.

The thing about programming using the asynchronous capabilities is that your program has to keep track of connection handles and to return to them and check for output. A bit more work on your part, but it allows the program to go "do other work" while waiting on a response.

© IBM Corporation 2010



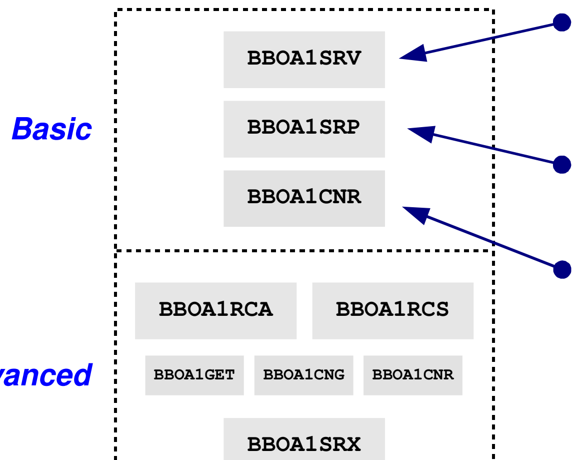
One of the parameters on the BBOA1REG API for registration was minconn and maxconn. Those set the minimum connections and maximum connections permitted across that particular registration.

The temptation is to think that "more is better," but that's not always the case. There are other limiting settings in the WAS environment. And there's no purpose in allocating a pile of connections if your program doesn't make use of them. There is a set of MODIFY DISPLAY commands that may be used to display the connection statistics for a registration.

The point of this chart is to let you know there is a "connection pool" at work, and the min and max is set at registration time. If you're using asynchronous features of the APIs, these connections have connection handles, and the advanced APIs have parameters to specify a given connection using its handle to pull messages back. This is what we meant earlier by having to "keep track" of things in your program.

Host a Service ... Establishing "I'm Listening" State

In order for an outbound call to work, the external address space has to put itself in a state where it can accept the call. That's called "hosting a service."



"Host a Service"
BBOA1SRV is a "basic" API. It ...
- Gets a connection from connection pool
- Puts program in listen state
- Holds program control until request received

"Send Response"
BBOA1SRP is used to respond back when the call comes from WAS

"Release Connection"
BBOA1CNR is used to release a connection. Since the basic API BBOA1SRV gets a connection each time, this is how you release it

CICS Link Server Task was doing this under the covers, which is how it shields your CICS programs from the APIs

Advanced APIs allow you operate asynchronous, maintain multiple connections, and multi-thread over them as needed

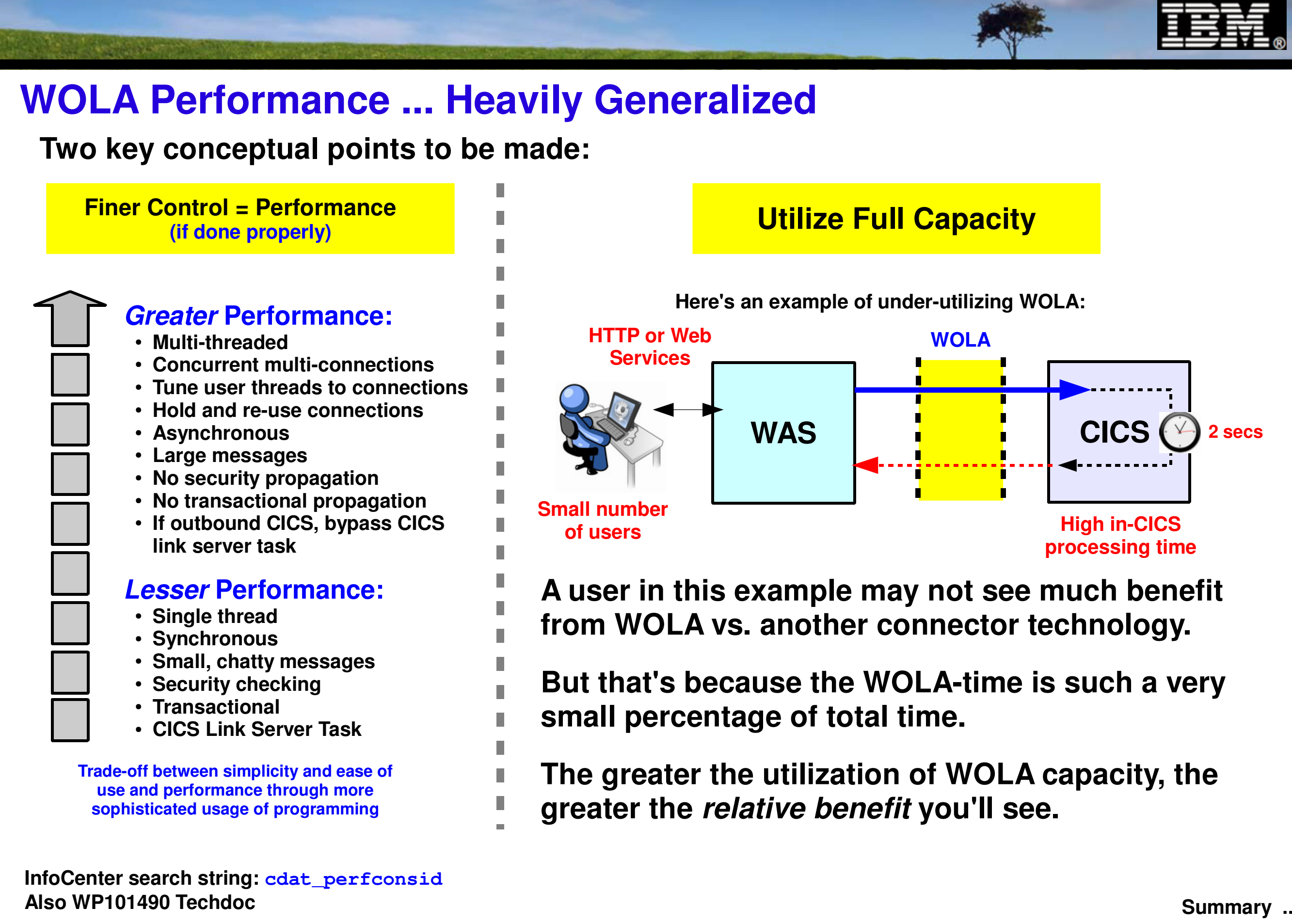
WOLA Performance ...

42 IBM Advanced Technical Skills © 2010 IBM Corporation

This chart shows the outbound APIs broken down by basic and advanced. Unlike inbound where a single API was sufficient, to "host a service" requires at minimum three (not including the registration and unregistration).

When the BBOA1SRV API is used it carries with it a name. This is what your Java program would specify on the `setServiceName()` method. Once BBOA1SRV is invoked, it waits for something in WAS to invoke it. With BBOA1SRV that operates synchronously, which means that thread is tied up until something comes in.

The advanced APIs provide an asynchronous variant of this -- BBOA1RCA or BBOA1RCS. Those "host a service" and return the thread back to your program. But then you have to come back with BBOA1GET to see if something has arrived. Greater control; greater performance.



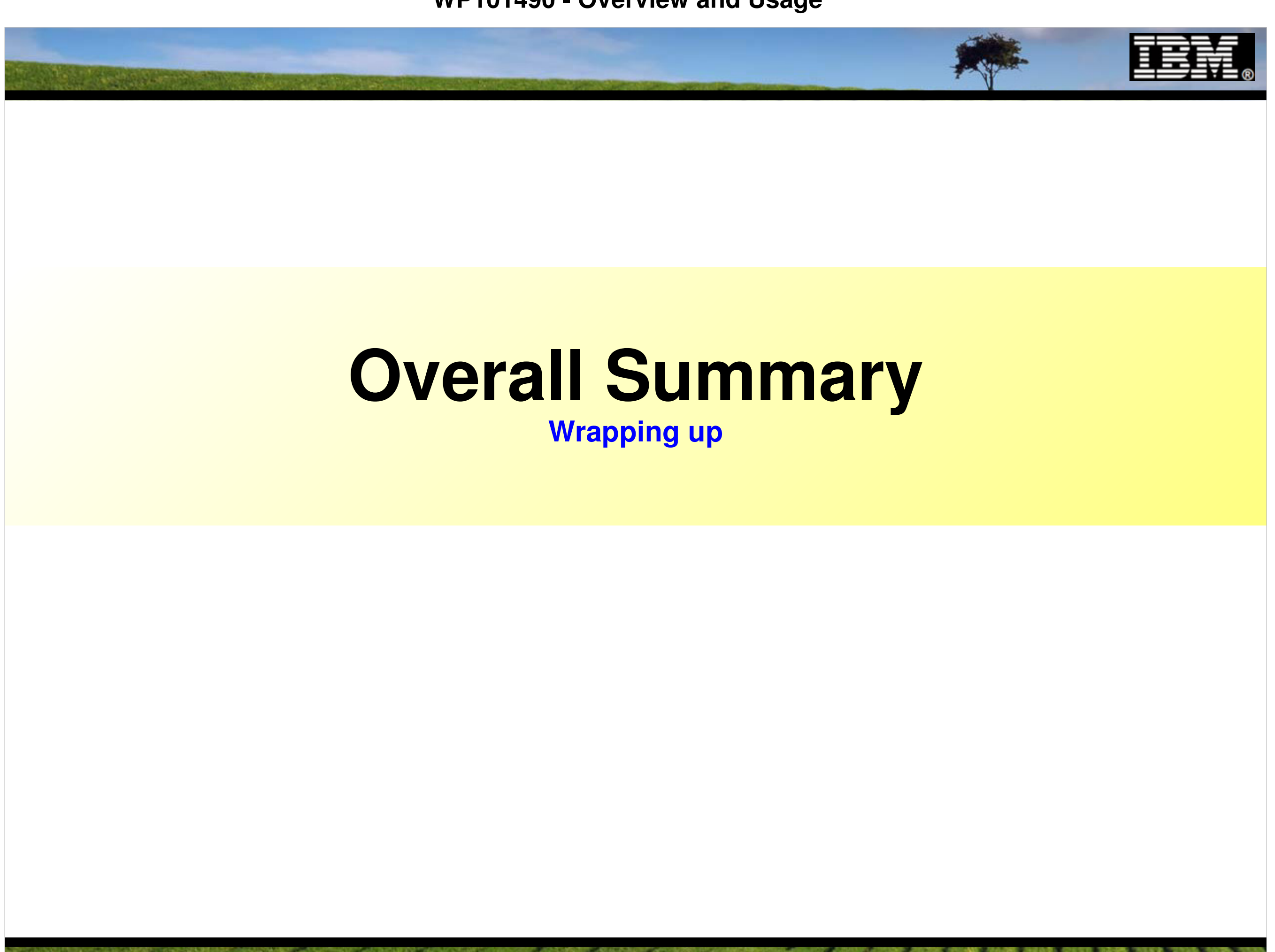
Now we get to the end of our presentation, and we offer a few comments about maximizing WOLA performance. This is a heavily generalized chart. The InfoCenter has more details.

The left-hand side of this chart is saying that there's a tradeoff between simplicity and performance. Simplicity comes by having WOLA functions make assumptions and remove from you the effort to do lower-level mundane things. As you desire to squeeze more and more performance out of a system, you start to take on more responsibility for coding (this is the asynchronous consideration), holding and re-using threads, bypassing the Link Server Task (or OTMA), and minimizing security checking, and having multiple threads of activity going at once.

The right had part of this chart is saying that you really see the benefits of WOLA when WOLA has a chance to run at near full capacity. By that we mean this -- we've seen several cases where someone does a single invocation test of, say, Web Services vs. WOLA. The stopwatch doesn't show much delta between those two for the single invocation. In both cases the utilization is negligible and it's hard to measure the differences.

Another case we've seen is where the time spent in the backend (CICS, or IMS, or batch or whatever) is relatively high while the transport time between WAS and that backend is relatively low. In that case the WOLA pipe remains relatively underutilized and the round-trip time is comprised mostly of the backend processing. If the throughput is gated by the limitation of the backend, then WOLA itself won't show much delta over other mechanisms.

For best results, use the pipe to the max.


Overall Summary
Wrapping up
44
IBM Advanced Technical Skills
© 2010 IBM Corporation

Overall Summary

Functionality

- Cross-memory single-LPAR byte area low-overhead exchange mechanism
- Inbound and outbound; CICS, IMS, Batch, USS and ALCS

Applicability

- Very well suited for inbound to WAS where other solutions may impose unacceptable overhead
- Excellent solution for high-speed batch interchanges

Programming

- Non-Java side: C/C++, COBOL, High-Level Assembler, PL/I
- Native APIs used as illustrated earlier and in WP101490 Techdoc
- Java side: code to CCI methods of supplied JCA adapter

Security

- Security propagation inbound and outbound is possible, depending on the case (see summaries)
- Region ID or Thread ID, inbound/outbound with CICS
- Thread ID in and out for IMS

Transaction

- Two-phase commit inbound and outbound WAS/CICS using RRS as syncpoint coordinator

Performance

- "Out of the box" basics provides very good performance
- Potential exists to tune even further using programming primitives as illustrated earlier
- WOLA will show greater and greater relative performance to other technologies the more you utilize the capacity of the WOLA connections

 © 2010 IBM Corporation

Our final chart summarizes the discussion.

© IBM Corporation 2010
Version Date: 11/12/12
WP101490 at ibm.com/support/techdocs

Document Change History

September 12, 2010	Original document
September 11, 2012	Updated the IMS section to reflect TX assertion IMS into WAS
November 12, 2012	Updated the IMS section to reflect TX assertion WAS into IMS over OTMA.

© 2010 IBM Corporation

End of Document

© IBM Corporation 2010