WebSphere Application Server for z/OS 7.0.0.4

A Brief Introduction to WebSphere for z/OS Optimized Local Adapters

(Sometimes referred to "WOLA" or "OLA" or "Optimized Local Adapters")

Version Date: July 2, 2009
See "Document Change History" on page 30 for a description of the changes in this version of the document

Written and provided by the WebSphere Application Server for z/OS team at the
IBM Washington Systems Center

Many thanks to **Jim Mulvey**, **Timothy Kaczynski**, **Tim Spewak**, **Dave Follis** and **Colette Manoni** of the WebSphere Application Server for z/OS development team.

Many thanks to **Leigh Compton** and **Dennis Weiand** of the ATS CICS Support team. Their help with the WOLA testing work was invaluable.

Also to **Mary Astley**, **Ken Hain**, **Jennie Liang**, **Glenn Materia, Brian Pierce** and **Dennis McDonald** of the Washington Systems Center Benchmark and Performance Testing team.

The WebSphere Application Server for z/OS support team at the Washington Systems Center consists of: **John Hutchinson**, **Mike Kearney**, **Louis Wilen**, **Lee-Win Tai**, **Steve Matulevich**, **Mike Loos** and **Don Bagwell**.

Mike Cox, Distinguished Engineer, serves as technical consultant and advisor for all of our activities. We could not do what we do without him.

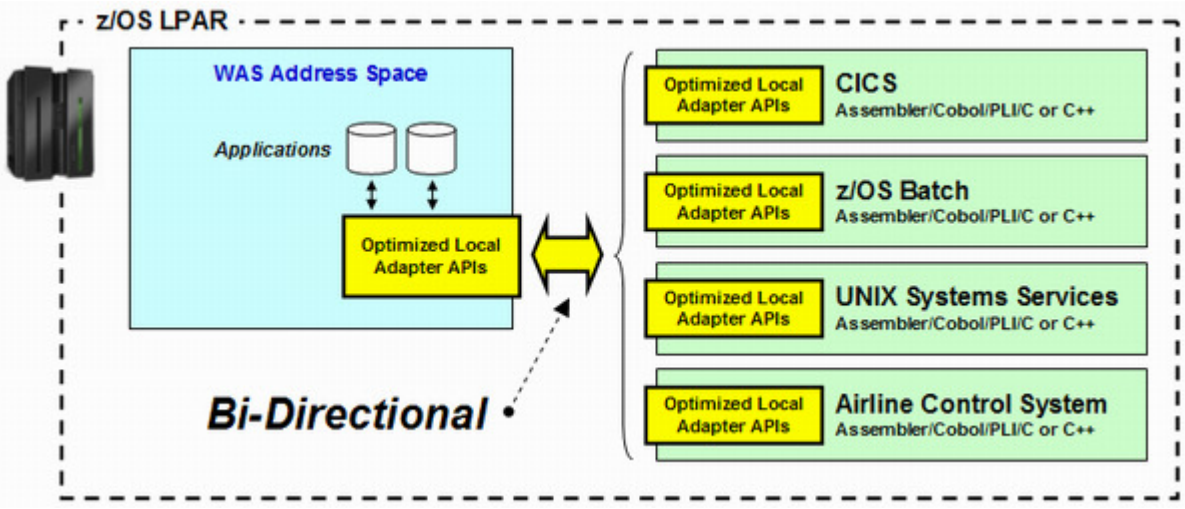
For questions or comments regarding this document, send e-mail to Don Bagwell at dbagwell@us.ibm.com

Table of Contents

Executive Overview

"Optimized Local Adapters" is a new function provided with WebSphere Application Server for z/OS maintenance level 7.0.0.4, released May 19th, 2009.

It is a new method of cross-memory local communications between WebSphere Application Server for z/OS and external address spaces such as CICS, batch programs, Unix Systems Services (USS) programs, and Airlines Line Control (ALCS) programs:



The new function is **bi-directional** – from WAS z/OS to the external address space, *or from the external address space into WAS z/OS*.

The key advantages of this new function can be summarized as:

- Very efficient cross-memory transfer from WAS to the external address space, or from the external address space into WAS
- Bi-directional capability – you can leverage WAS EJB assets as local services from external address spaces such as CICS or batch programs.
- Security propagation – from WAS you can flow the user ID, the servant ID, or the EJB role ID into the external address space; or from the external address space you can flow the client ID or, in the case of CICS, the CICS region ID or the CICS task userid.
- Transaction propagation – When operating from CICS into WAS, WAS can participate in a CICS unit of work for two-phase commit processing. However, in the initial release of WOLA in Version 7.0.0.4, transaction is not supported for flows from WAS into CICS, otherwise known as "outbound" from WAS.

In summary, the Optimized Local Adapter represents a way to link WebSphere Application Server for z/OS and external address spaces in an optimized, high-speed, cross-memory bi-directional manner. Leverage your assets to their fullest advantage. Realize high degrees of scalability and throughput. Capitalize on the extreme security of operations within the z/OS operating system.

That is what the WebSphere Application Server for z/OS optimized local adapter is all about.

Note: This paper is designed to be an introduction to WOLA, but certainly *not* a complete source for reference information, or a a complete activity checklist for enabling and using the new function. The WebSphere z/OS InfoCenter has been updated to provided those details. We offer pointers to specific articles in the InfoCenter throughout this document.

Questions and Answers

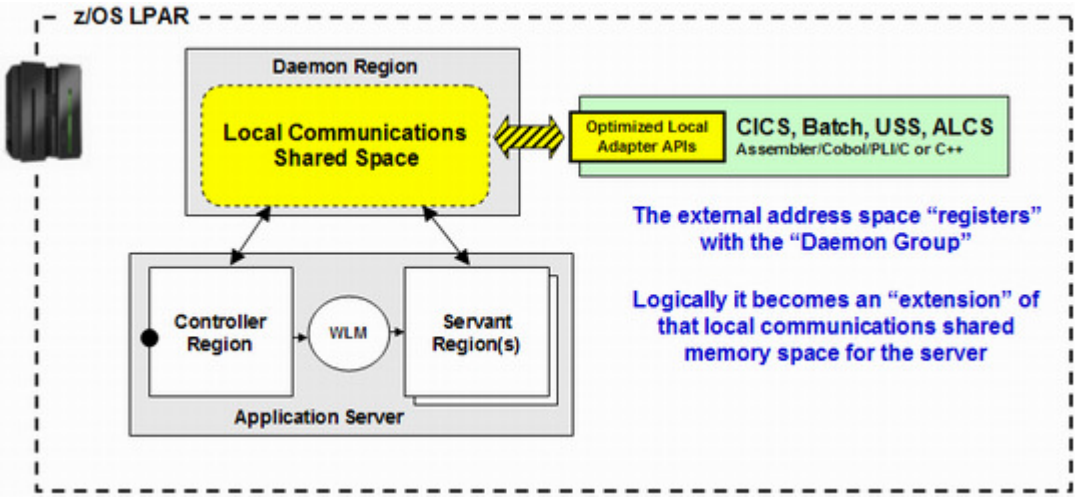
1. *What exactly is this thing?*

The WebSphere Application Server for z/OS optimized local adapters (WOLA) is a relatively low-level communication mechanism that allows cross-address space communications from WebSphere outbound, *and* from external address spaces inbound to WebSphere.

WebSphere Application Server for z/OS already has a cross-memory communication structure used internally. It makes use of the Daemon server's shared memory that allows address spaces within that cell on that LPAR to communicate cross-memory.

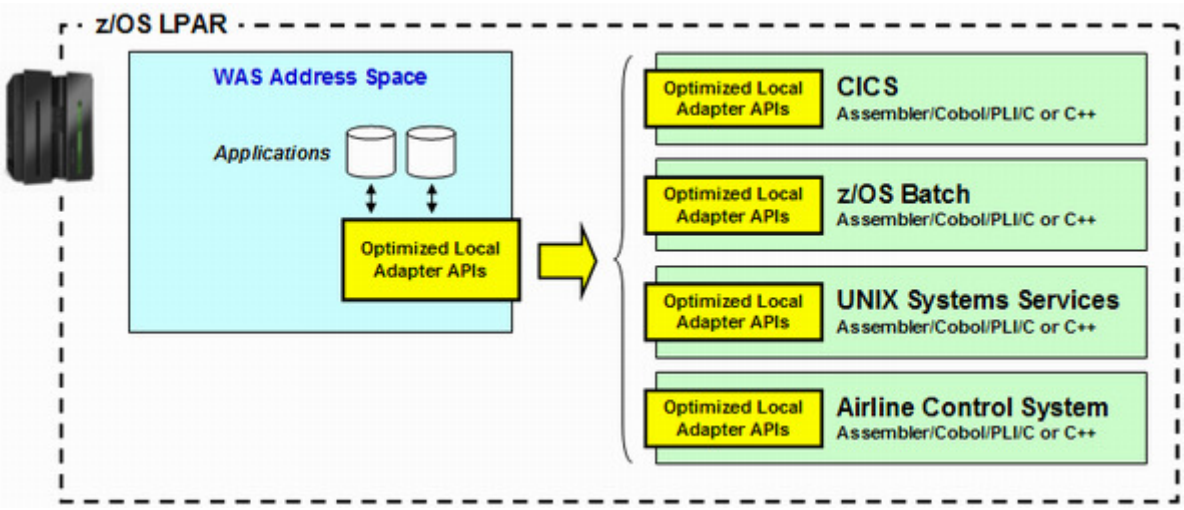
Think of WOLA as an *extension* to that capability. The extension is the ability of external address spaces to participate just as address spaces inside the cell can.

Here's a picture that provides a *conceptual* representation of this:



Note: Focusing on the Daemon's role in this will prove useful when you see how an external address space participates using WOLA. One of the first steps is to "register" to the Daemon group and provide information about what node and what server the cross-memory communications is intended for.

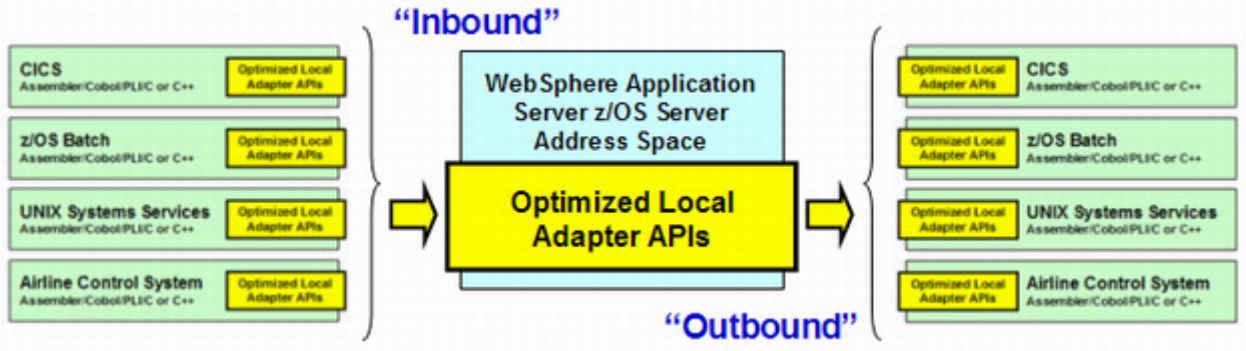
The outbound support (WAS to external address space) is the easiest to grasp because that's how we normally think of WAS communications: from WAS out to data such as CICS. This picture represents this:



You'll notice there's a box in the WAS address space labeled "Optimized Local Adapter APIs." That is provided with a standard JCA resource adapter named `ola.rar`. That resource adapter is installed like any other, and Java applications in WAS communicate using the standard JCA application interface.

On the external address space side the support is provided in the form of module libraries that you make available through `STEPLIB` or `LNKLIST` (or, in the case of CICS, `DFHRPL`).

What about *inbound* to WAS? Well ... just flip the picture. The same things apply. So if we combine inbound and outbound on the same picture, we get:



There is much we haven't touched on yet. But we'll get to a bit more detail as this document unfolds.

Note: The WebSphere Application Server for z/OS InfoCenter is an excellent resource for detailed information about this new function:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp`

2. What are the basic WebSphere z/OS requirements for this?

Briefly:

- WebSphere Application Server for z/OS Version 7.0.0.4
- A WebSphere Application Server for z/OS environment built (either Standalone or ND)
- That WAS z/OS node is enabled for WOLA (see "What's involved with enabling WOLA in a node?" on page 20)
- Servers that make use of WOLA must operate in 64-bit mode, which is the default mode for WAS V7 servers[1].

3. Don't we already have local adapters? What was the motivation behind this?

We do. They are very good. They should still be part of your architectural toolkit.

The original spark behind this was the desire on the part of some customers to provide a way to leverage EJB assets in WAS z/OS from z/OS *batch programs*. Solutions existed, but each had its limitations[2]:

[1] There has been considerable work done to make the performance of 64-bit mode comparable to 31-bit. One such mechanism is something called "compressed references." Find a write-up on this technology with supporting benchmark data at: `ftp://ftp.software.ibm.com/software/webserver/appserv/was/WAS_V7_64-bit_performance.pdf`

[2] Please do not construe this as our suggesting WOLA is "better" than these technologies in *every* way. All technologies have their particular limitations, including WOLA (see "What are the limitations?" on page 6). We cite these limitations simply to explain the origins of WOLA.

MQ	Very good for asynchronous transfer. But isn't designed for transactional capability.
RMI	Proven technology, but it implies a Java Virtual Machine, and Java code. Quite possible, particularly with JZOS[3],but that was seen as additional complexity compared to a simple non-Java batch program interaction.
Web Services	Ubiquitous open standard, but generally seen as too limiting for very high-volume transaction rates.

So the developers in Poughkeepsie, New York investigated the concept of providing a cross-memory mechanism to come *into* WAS from an external batch program. The idea of extending the internal WebSphere Daemon cross-memory mechanism came from that.

Limiting it to *just* inbound was seen as an incomplete solution, so the design called for a bi-directional mechanism. Hence the *inbound* or *outbound* design described earlier.

The developers could have avoided coding to CICS in this first round, but that too was seen as making WOLA an incomplete solution.

The CICS support of WOLA naturally brings up a question of positioning against CTG ...

4. Does this eliminate the use of CICS Transaction Gateway (CTG)?

Not at all. CICS Transaction Gateway has many important roles to play in an enterprise architecture. It does things WOLA isn't designed to do:

- CTG provides the ability to access CICS resources from WebSphere Application Server in another LPAR, or another platform. WOLA is designed to be same-LPAR cross-memory only.
- CTG plays an important role in providing a highly available architecture[4]. WOLA is not capable of routing requests to different CICS regions on different LPARs in the event of a region loss locally.
- From WebSphere into CICS, the CTG provides the ability for CICS to participate in two-phase commit processing. With WOLA, in the initial release in Version 7.0.0.4, only "synconreturn" is supported for outbound from WAS into CICS.

> **Note:** If you reverse that flow – that is, use WOLA to go from CICS into WAS – then WebSphere **can** participate as a unit of work in CICS two phase commit processing.
>
> The current restriction is *just* on outbound, WAS into CICS.

In summary, WOLA is *not* designed to replace CTG, but to *complement* it. WOLA excels when bi-direction high-speed communication is needed between WebSphere Application Server and CICS.

[3] JZOS is a batch Java Virtual Machine invocation mechanism supplied with the z/OS operating system. It was developed by another company, and the technology was acquired by IBM in 2005. JZOS overcomes many of the limitations of JVM invocation using `BPXBATCH`.

[4] As stated elsewhere, a great deal goes into a comprehensive high availability architecture. The connectivity from application server to data source is but one element. Therefore, CICS Transaction Gateway does not *all by itself* provide high availability; rather, it's an element in an *overall design*. Another common high availability technique with CICS is to use CICSPlex with terminal owning regions that load balance to multiple application owning regions in the Plex. WOLA could be the connection between WAS and one of those TORs. But because of WOLA's cross-memory design, it can't reconnect to a different TOR elsewhere in the Sysplex. There are CTG topology designs that do permit this.

5. What's the carrot here ... why should I consider this?

Let's explore the basic value statements of this new function:

- *Cross-memory inbound exploitation of WAS Java assets*

  There is no other cross-memory method of invoking EJB assets in WebSphere Application Server for z/OS that also supplies transactional context. That is probably the single most striking differentiator of this new function.

  An easy-to-overlook element of this is the fact this inbound access is supported using batch languages such as COBOL, C/C++, PL/I or High Level Assembler. Those are highly efficient compiled languages. No separate JVM is required to use WOLA inbound to WAS.

- *Performance and throughput*

  Because of the efficiency of the cross-memory transfer and the low latency involved, the performance and throughput profile of this new function is very good. We are performing benchmarking studies to validate the improvements to expect. The numbers are not yet ready for publication. We'll make those numbers available as soon as we can be assured we have them properly tested and measured.

Important: We are very careful to remind you that WOLA is not a universal solution for all situations. *By no means* should you consider this a total replacement for any existing connectivity technology, such as MQ , or CICS Transaction Gateway, or Web Services. Each has its place, as does WOLA. Evaluate each situation based on the needs of the business.

6. What are the limitations?

Let's review some of the obvious and not-so-obvious limitations of this WOLA function:

- *It is limited to cross-memory communications within a single z/OS operating system instance:*

  It can't be used to communicate from one LPAR to another in a Sysplex.

  It can't be used to communicate from Linux for System z to z/OS.

  It can't be used to communicate from WebSphere Application Server on a distributed platform to an address space on z/OS.

  Note: That's the price one pays to do cross-memory communications. Gain the speed and efficiency, lose the flexibility to talk over a routed network.

- *There is no ability to recover from an outage of its communication partner and re-establish communication to another partner:*

  That tends to limit the high availability options *at that level of the architecture*. However, it's very important to realize that high available can – and should – be a total architecture evaluation and not simply the linkage from application to data.

  By contrast, using the CTG resource adapter and a TCP connection to a CTG Gateway Daemon provides the ability to exploit Sysplex Distributor so that if one Gateway Daemon goes down a re-connection will find other Gateway Daemons. Or, by contrast, a client mode MQ connection can go to any Queue Manager in a queue sharing group.

  Note: Again, a conscious trade-off of connectivity flexibility for optimized throughput. And as stated, high availability architectures involve many "layers" – from way down at the Parallel Sysplex level, up through data and application systems, and on up to the network layer. High availability can be achieved with WOLA, but by exploiting solutions at the higher levels.

- *There's a current limitation on outbound support to CICS – synconreturn only*

  In the initial release of WOLA in Version 7.0.0.4, using WOLA to go from WAS into CICS, we are limited to "synconreturn" only. There is a work item in place to address this in a future maintenance release.

  However, because of the way WOLA is implemented in CICS (as a Task Related User Exit, or TRUE), eventually we'll run into *another* known limitation. That is, outbound calls from WAS to CICS will only be able to use "last participant" (or "synclevel 1" in CICS parlance) transaction scope.

  At 7.0.0.4 we're limited to synconreturn. When that's addressed, we'll then be limited to "last participant" because of a restriction in the architecture of the CICS TRUE.

  Important! *Inbound* from CICS to WAS is capable of allowing WebSphere to participate in a CICS unit of work, with full two-phase commit support.

7. Is this transparent to the application?

We need to answer this question *very carefully*. There are really two sub-questions:

- *Is it necessary to modify *__every__* application that seeks to use WOLA?*

  The short answer is "No." There are many scenarios where existing and new applications can take advantage of WOLA *without* specifically writing to the WOLA APIs. That true of both WAS EJBs as well as CICS programs.

  CICS programs are what people ask about most often. And the answer for CICS is – *existing CICS programs do not necessarily have to be modified to use WOLA to interact with WAS.*

  Note: The word "necessarily" is there because for a CICS program it depends on the nature and structure of the existing program. One that is well structured and can accept COMMAREA or Containers can remain unchanged. The WOLA piece can remain separate from the existing program, and the existing programs may simply be called.

- *Is it necessary to write or modify *__some__* applications to use WOLA?*

  The short answer is "Possibly, yes." Depending on the scenario, some custom code exploiting the WOLA APIs is going to be needed. For example, a C/C++ *batch* program intending to use WOLA to use a WAS-hosted EJB will need to use the WOLA APIs to get into WAS. For CICS, if the desire is to call into WAS and drive EJBs, then *some* CICS program exploiting the WOLA APIs will be needed. But again, properly structured, other existing CICS programs can remain unchanged ... they call the WOLA-custom program using COMMAREA or Containers.

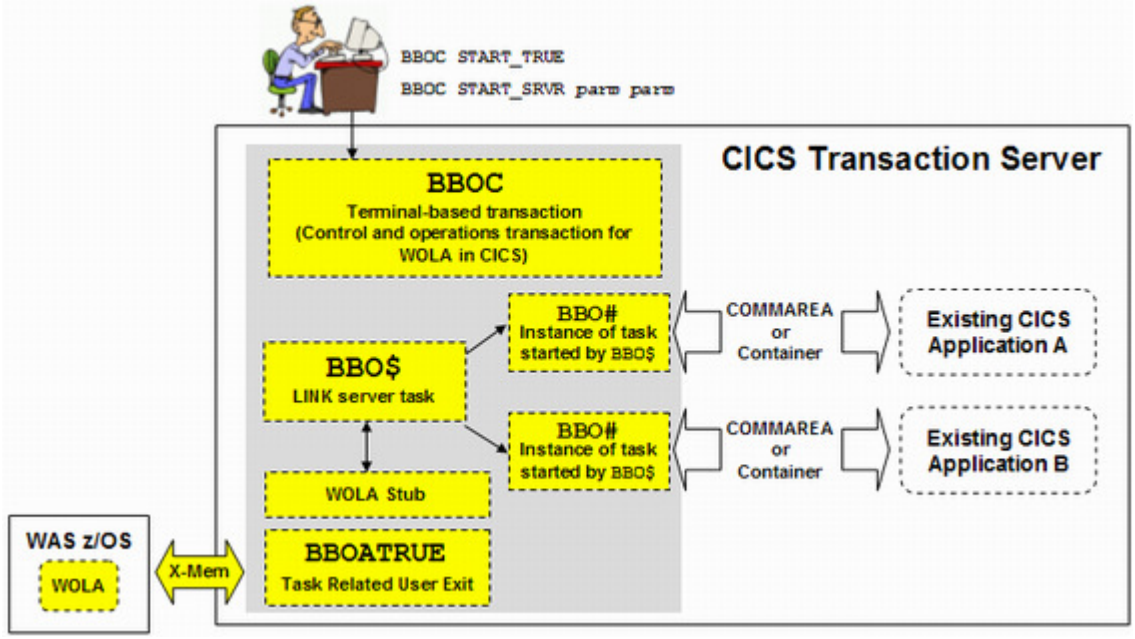
  If all you have is outbound from WAS to CICS, and the CICS programs are well structured and can be called using COMMAREA or Containers, then you can operate with the WOLA-supplied `BBO$` and `BBO#` transactions. The InfoCenter has more. Search on `BBO$`.

Let's back up and organize things into categories of usage:

WOLA and CICS

We touched on this in the two questions above. Why the answers are what they are is based on how WOLA is implemented in CICS. The brief explanation is that WOLA is implemented as a Task Related User Exit (TRUE), with a supplied control transaction (`BBOC`), two LINK server transactions (`BBO$` and `BBO#`), and some stub code.

That provides a WOLA framework in CICS that can be used by WAS to access CICS programs without *necessarily* having to modify the CICS programs at all[5]. Here's a high-level picture that represents this:



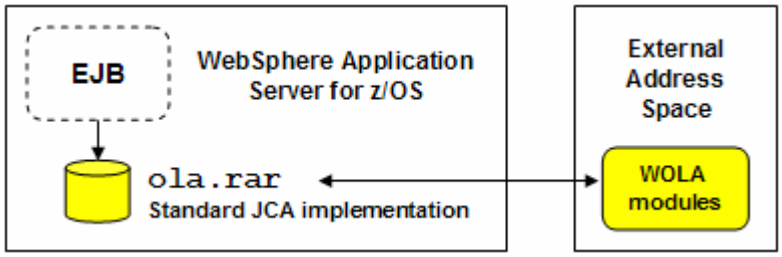
There's a lot more detail behind that picture, of course. But that pictures helps you visualize how existing CICS programs can remain unchanged *and* take advantage of WOLA.

For CICS programs looking to initiate calls *into* WAS, then a program will need to be written that exploits the WOLA APIs to drive the EJBs. Some of the samples illustrate how this can be done. (See "What other samples are provided?" on page 23.) Again, that program can be called by other CICS programs using COMMAREA or Containers. Properly architected, one WOLA-enabled CICS program can be used by many other CICS programs to access EJBs in WAS.

See "How is the CICS support implemented?" on page 16 for more.

WOLA and WAS z/OS

From the perspective of applications in WAS, they access the underlying WOLA code through a supplied standard JCA resource adapter. That's how an application in WAS is going to go out through WOLA, or be accessed from outside through WOLA:



[5] The WOLA API does not provide access to all the containers in a channel, just indexed records. This is a difference between the usage of CICS containers in other environments, such as in the CTG JCA or inside CICS. Since containers *may* be added or deleted from the channel by *other* programs linked to in the application, it may require changes to CICS applications to allow re-use with WOLA.

Note: That picture is somewhat simplified. There is also a WAS-supplied native module that's called by the JCA adapter. That's what does the actual cross-memory linkage to batch and CICS. That native module is enabled with the `olaInstall.sh` script you'll read about later. Once enabled, it becomes part of the node's structure like any other WAS module. Generally speaking, you don't have to think about it ... it's just there.

The methods on that resource adapter are the same as supplied on any JCA resource adapter.

The JCA objects that are customized for this adapter are as follows:

- ConnectionSpec
- InteractionSpec
- Record I/O

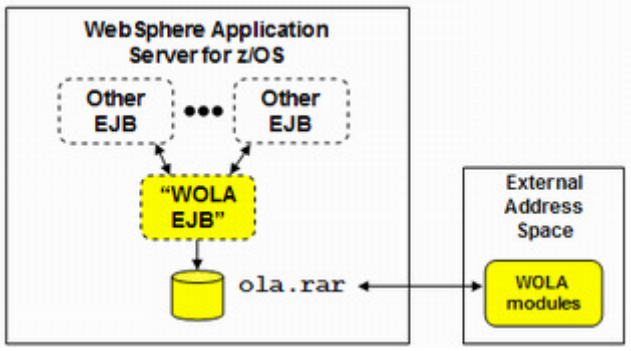
The InfoCenter has a very good write-up on what an EJB[6] must implement to make use of the WOLA JCA adapter:

> The assembly tools are shipped with the WebSphere Application Server for z/OS and contain the Java archive (JAR) file that contains the package, `com.ibm.websphere.ola`, that you will need to identify your enterprise bean as a potential target of an optimized local adapter call.
>
> This package contains the ExecuteHome and Execute classes that hold the abstract interfaces that are needed to call the adapter. You must create a home and remote interface for your enterprise bean that implements `com.ibm.websphere.ola.ExecuteHome` for the EJB home interface and `com.ibm.websphere.ola.Execute` for the remote interface.
>
> The enterprise bean that you want to invoke from an external address space must include a method called `execute` that accepts a byte array as input and returns a byte array as output. This is the method that you will receive control in when an external address space uses one of the adapter API calls such as Invoke or Send Request.
>
> The `execute` method is defined on the remote interface, `com.ibm.websphere.ola.Execute` and contains the business logic for the application.

Search on `tdat_useola_in_step2.html` to go straight to that article.

The message here is that the EJB used to interface to the WOLA JCA adapter must be written to provide certain things. So in that sense WOLA is not "transparent" to EJBs.

Keep in mind that your application structure could include a WOLA-specific EJB that acts as a kind of "WOLA-access service" for other EJBs:



So it's not necessary for *every* EJB to be specifically written (or re-written) to conform to the WOLA requirements. But some EJBs will.

[6] For EJB3 you'll need a `@RemoteHome com.ibm.websphere.ola.ExecuteHome.class` annotation for the target stateless session bean. This is the EJB3 way to declare a Remote Home and Remote interface. For EJB 2.1, this is done the old way, within the deployment descriptor.

WOLA and Native Batch

Batch programs – including USS and ALCS programs – will require custom code to exploit the WOLA APIs to access WAS EJBs. The sample `OLACC01` is the simplest example of this. See "What's the simplest way to validate the enablement of WOLA?" on page 22 for an illustration of this very simple sample batch program.

WOLA and Java Batch?

The short answer is "No, not *directly*."

> **Important:** Let's be clear that WOLA is certainly supported for Java EJBs running in WebSphere Application Server for z/OS, provided the node has been enabled for WOLA. That's one of the fundamental purposes of WOLA

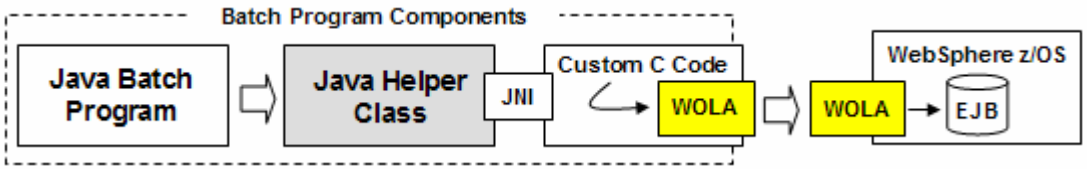
The question *here* is for for Java programs running *outside* the WebSphere Application Server for z/OS environment. Three such environments come to mind:

- Java batch invoked with BPXBATCH
- Java batch invoked with JZOS
- Java executed within a CICS region JVM environment

The essential issue here is that WOLA is a native code implementation. It operates at a level fairly low in the WebSphere z/OS architectural model. That's why it is so efficient.

Because WOLA is implemented in native code, to go from Java to WOLA requires a JNI call and a switch to the WOLA code. The supplied WOLA JCA adapter does just that.

Coming from a Java program *outside* of WebSphere z/OS would require some Java helper classes that would make the JNI call to C/C++ code which then invokes the WOLA APIs:



That is certainly doable. But the initial implementation of WOLA does not provide those Java helper class files. You could certainly write them with sufficient knowledge of JNI calls.

If WOLA is not used, then another way for a Java program on z/OS to access an EJB in WebSphere z/OS is to make us of the local IIOP classes. That can be a fairly efficient mechanism.

8. *What kind of functions do the APIs provide?*

Here's a bitmap clip straight from the InfoCenter[7]:

- Register - BBOA1REG
- Unregister - BBOA1URG
- Connection Get - BBOA1CNG
- Connection Release - BBOA1CNR
- Send Request - BBOA1SRQ
- Send Response - BBOA1SRP
- Send Response Exception - BBOA1SRX
- Receive Request Any - BBOA1RCA
- Receive Request Specific - BBOA1RCS
- Receive Response Length - BBOA1RCL
- Get Message Data - BBOA1GET
- Invoke - BBOA1INV
- Host a Service - BBOA1SRV

In the InfoCenter each link jumps to a very detailed section on that particular API, including a map of the parameter values. For example, `BBOA1REG` shows this:

Table 1. BBOA1REG API syntax

API	Syntax
BBOA1REG	`BBOA1REG ( daemongroupname , nodename , servername , registername , minconn , maxconn , registerflags , rc, rsn )`

There's a description of each parameter, usage notes, and a detailed table of return and reason codes. It's an excellent reference source for detailed information.

We don't want to drill straight into the details of each and every API in a "Brief Overview" paper. As mentioned, the InfoCenter has an excellent detailed write-up.

But to give you a sense for this, let's explore a *few* of the APIs.

The most fundamental APIs: `BBOA1REG`, `BBOA1INV` and `BBOA1URG`

Before an external address space program can communicate with a WebSphere Application Server for z/OS application, it must *register* to the Daemon group for the target WAS server. That registration is done with the `BBOA1REG` API.

When all the work is done, the external address space program should then *unregister* from the Daemon group. That's done with the `BBOA1URG` API.

Between registering and unregistering the external address space program will want to *invoke* the target EJB in WAS. That's done with the `BBOA1INV` API.

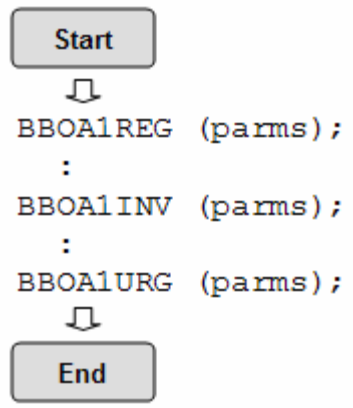
Here's a peek into one of the samples – `OLACC01` – which is a simple C/C++ batch program that invokes the sample EJB in WebSphere and quits[8].

> **Note:** `OLACC01` is the simplest sample. Use it to do your most basic validation of the WOLA installation. See "What's the simplest way to validate the enablement of WOLA?" on page 22 for a discussion of how to use `OLACC01` for that purpose.

Inside the `OLACC01` sample the API usage flow is like this:

[7] Search on the string `cdat_olaapis.html` to go straight to the page with the API details.

[8] See "What other samples are provided?" on page 23.



Let's explore that a bit:

- The `BBOA1REG` API is used to register to the Daemon group and pass in the node and server name to connect to. A unique registration name is supplied, which is what the Daemon uses to keep separate what may be multiple registrations[9] into it from outside address spaces.

  Note: Suppose the WAS cell was completely down when you invoked that API. What would you see? You'd see RC=12, RSN=10, which the InfoCenter informs us means, "Unable to locate the selected WebSphere Application Server daemon group." In other words, the Daemon isn't up ... therefore, the connection to its shared memory isn't possible.

  We offer that just to give you a feel for how this works – when it works and when it does *not* work. Again, consult the InfoCenter, which has all the details.

- The `BBOA1INV` API is used to invoke the EJB in WAS, using the home interface JNDI name as the "service name" of the EJB. The data is passed over.

  Note: Suppose the JNDI name you supplied had a typo error in it. The EJB you intended to invoke was there and active, but because of your typo it couldn't invoke the EJB. What would you see?

  You'd see RC=8, RSN=34, which the InfoCenter informs us means, "The target service not found."

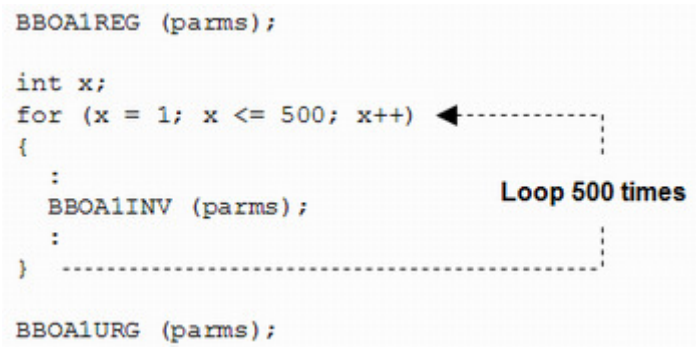
- The `BBOA1URG` API is used to un-register, or tear down the registration. The program exits.

Experimenting with `OLACC01` and making a modification

Those three APIs represent the *simplest possible* flow: register, invoke, then unregister and quit.

A simple variation on that would be a looping structure, which you could easily add to the `OLACC01` sample as part of early experimentation with the WOLA function:

[9] The z/OS MODIFY command `F <jobname>,DISPLAY,ADAPTER,DAEMONRGES` will display all the current registrations for a given server.



The key point here being there's a *single registration* (BBOA1REG), then 500 invocations of the EJB (BBOA1INV), then *a single unregister upon exiting the loop* (BBOA1URG).

Hosting a service, and sending a response

The OLACC01 sample we just looked at was entirely *inbound to WAS*. That is, the C/C++ program didn't have to worry about waiting and listening for someone to call it. It invoked the EJB and processed the response.

Let's say you want to code a program that waits for an EJB to call it. What API do you use?

The sample OLACC02 illustrates the use of the BBOA1SRV ("host a service") and BBOA1SRP ("send a response") APIs.

Note: OLACC02 is a bit more complex than OLACC01, but not terribly so. It'll take a bit more concentrated thinking to follow the logic and usage of OLACC02, but in doing so you'll get a better appreciation of more of the APIs. OLACC01 is the barest of the bare. OLACC02 is the next level of detail

Other APIs

The other APIs are a bit more detailed. If you're a programmer and are familiar with using communication-oriented APIs, these should prove fairly easy to master.

9. What programming languages does WOLA support?

From the external address space, these callable services can be used from the following native programming languages:

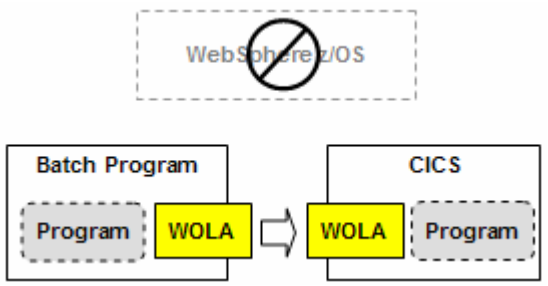
- Cobol
- C/C++
- PL/I
- High Level Assembler

Those are very fast and efficient compiled languages. WOLA gives you the ability to use those to leverage EJB assets in WebSphere z/OS, or have those EJBs leverage your native code.

In WebSphere Application Server, of course, the language is Java. We addressed the question of Java *batch* under "WOLA and Java Batch?" on page 12.

10. Can I use WOLA to go from Batch to CICS without WebSphere z/OS?

Here's a picture of what's being asked:



No. WOLA is, fundamentally, an extension of the WebSphere Application Server for z/OS runtime environment. It is intended to communicate from WAS z/OS out to supported external address spaces, or from those address spaces back into WAS z/OS.

Going from one external address space to another using just WOLA (but no WAS z/OS) is not possible.

The reason is goes back to the picture we showed earlier, which illustrated how the WAS Daemon shared space plays an important role in coordinating communications using WOLA. One of the very first things that must be done is to register into the Daemon group. If WAS z/OS is not part of the picture, then no such registration is possible.

11. How is the CICS support implemented?

What follows is straight out of the InfoCenter[10]. If you're not comfortable with CICS concepts, don't worry: take this to your CICS system programmer and they'll understand what this means:

> The adapter is designed to execute in a CICS region as a resource manager. In CICS, the Task Related User Exit (TRUE) is the primary vehicle used by resource providers. TRUE support provides the boundary between the CICS application threads and the external resource manager threads. Currently, DB2, MQ, and TCPIP sockets execute in CICS using the TRUE support. The optimized local adapters support TRUE.
>
> Applications that run under CICS and exploit the optimized local adapter APIs do so by calling the provided stub routines. The stub routines invoke the CICS resource manager interface module, passing it the name of the optimized local adapter TRUE routine and the parameters specific to each API. CICS dispatches the TRUE on one of the CICS-maintained OPENAPI TCBs and executes until the API call completes. The call then goes back to CICS with the output parameters. The CICS TRUE support also provides for notification over transaction boundaries, such as when the application ends normally, abends, or issues an explicit SYNCPOINT call to CICS. For details on how this support is used for propagation of transaction context to WebSphere Application Server, and two-phase commit, refer to the section below on Propagating Transactions. Additionally, for CICS, a Program List Table Post-Initialization (PLTPI) program is provided, that can be used to automatically start the TRUE program during CICS startup. If you do not use the PLTPI, a CICS transaction, BBOC, is provided. This transaction can be used to start, stop, enable, and disable tracing for the TRUE module.
>
> The BBOC transaction is the WebSphere control, or operations transaction, for the adapters support under CICS. It is used to enable and start the WebSphere Application Server TRUE, as well as to set tracing levels for debugging the APIs and WebSphere Application Server-interfacing code. BBOC is also used for setting up registrations and unregistering, as well as starting and stopping WebSphere Application Server Link server tasks in CICS. These server tasks provide support for invoking existing CICS programs and passing data with COMMAREAs or containers with the input parameters over the adapters. This is a CICS terminal-based transaction that can be issued on a 3270 terminal or from a sequential, SDSCI-type, terminal.

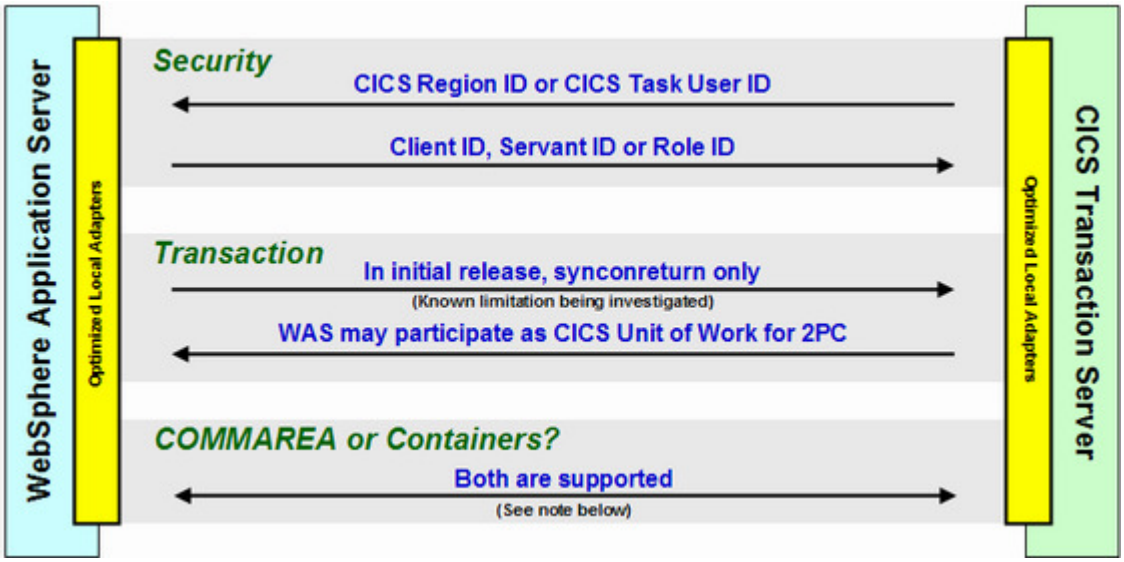
[10] Keyword search in V7 InfoCenter: `rdat_cics.html`

The CICS transaction, BBO$, is used for WebSphere Application Server outbound call support to CICS. It runs in the background as a non-terminal transaction and represents an instance of a server task that is started using operation, BBOC START_SRVR, for a user-specified register name and service name. This server task provides a program LINK invocation capability, or the ability to invoke an existing CICS program from WebSphere Application Server over the optimized local adapter APIs. The name of this transaction can be overridden by a user-supplied name that can be provided on the BBOC START_SRVR command (STX=xxxx). If this is done, the provided transaction name must be defined to CICS with the same program name and attributes as BBO$.

The CICS transaction, BBO#, is used for WebSphere Application Server outbound call support to CICS. It is a non-terminal transaction that represents an instance of a task that was started by the WebSphere Application Server adapters server task, BBO$, in order to do a program LINK invocation. The BBO$ transaction initiates a BBO# transaction for each CICS program link request from an application that is deployed on WebSphere Application Server.

12. What about security and transaction, in and out of CICS?

Here's a picture that *summarizes*[11] the support:



Note: WOLA's support for channels/containers is not quite as robust as CTG's. WOLA uses a default channel name that we have preset and have set/get methods that allow for selecting a 'request' container name, the 'response' container name and the types of these (byte/char). Then in WOLA's link server under CICS (the `BBO$` task) WOLA uses these to pass the request data in for the target program and pull the response data out to send back. WOLA does not currently support the MappedRecord interface that CTG provides today.

Depending on the CICS application's use of containers, that may imply modifications to use WOLA.

We have commented that the WOLA support in WAS z/OS is provided as a standard JCA adapter. While that is true, it should be noted that the APIs to access CICS are not *identical* between the WOLA JCA and the CTG JCA. One such difference is the ability to access all containers in a channel: CTG supports this; WOLA does not. WOLA supports bi-direction; CTG does not.

Again, we stress repeatedly that WOLA and CTG are *complementary* and **not** exclusive of one another.

[11] There's a bit more to this. There are API flags that need to be set, as well as `CBIND` class profile update. The InfoCenter covers the details of all that.

13. What about development tooling in support of using WOLA?

There is support provided to use tools such as Rational Application Developer with WOLA.

The following InfoCenter articles are available to assist in this area:

InfoCenter Article Title	*Keyword Search*
Using optimized local adapters for inbound support	`tdat_useola_out.html`
Using the optimized local adapter to connect to an application in an external address space from a WebSphere application	`tdat_connect2wasapp.html`
Optimized local adapters for z/OS APIs	`cdat_olaapis.html`
Accessing data using Java EE Connector Architecture connectors	`tdat_impjcaapi.html`

14. Is there an additional charge?

No. It's part of the WebSphere Application Server for z/OS Version 7 maintenance stream.

15. How is this packaged?

It's packaged as part of the 7.0.0.4 maintenance release, and there is no charge.

Configuration file system pre-enablement

When 7.0.0.4 is applied and `applyPTF.sh` runs for a node, you'll see the following in the configuration file systems for each node:

That does *not* mean WOLA is enabled and ready to go. That is merely the shell script that is used to enable the node. We'll cover what's involved with that under "What's involved with enabling WOLA in a node?" on page 20.

Notes:
- Enabling WOLA is a node by node thing.
- Not every node needs to be enabled for WOLA. Only those nodes with servers you wish to use WOLA.
- Geneally speaking the DMGR node does not need to be enabled for WOLA. One exception we've seen is where an application written to make use of the WOLA JCA adapter is installed with the EJBDeploy option selected. That implies a Java build process, which means the DMGR needs access to the `ola_apis.jar` file[12]. Enabling the DMGR node for WOLA provides that.

When you enable a node, the `olaInstall.sh` shell script creates some symbolic links in the configuration file system to link the node back to the SMP/E file system, where it picks up the WOLA files.

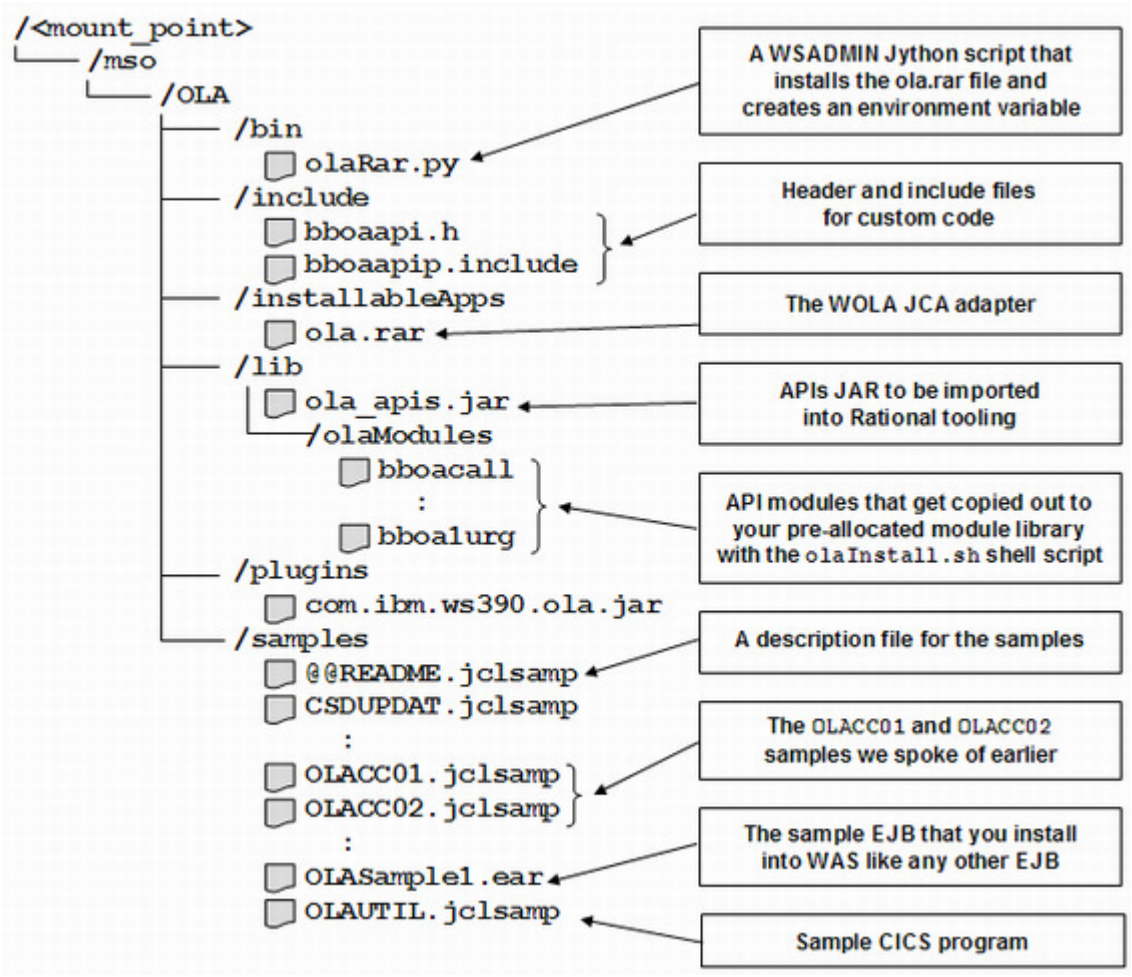
[12] If your EJBs are generated with deploy code in your tooling, then you would not need to worry about this. Importing the `ola_apis.jar` file into the tooling is needed to support development of EJBs that will use the WOLA JCA. That would provide the necessary access to the classes in `ola_apis.jar`. The resulting EAR could then be installed *without* the EJBDeploy option, and thus the DMGR node would not need to be enabled for WOLA.

Notes: If you're thinking, "What about 'intermediate symbolic links?', then ...

- The shell script will piggy-back on what the node currently has configured for the location of the SMP/E product file system. If intermediate symbolic links are used[13] then the shell script will automatically pick up and use the value of those intermediate symlinks.
- Bravo! Thinking about how this new function links back to the SMP/E product file system is a very subtle point. If you were thinking about it, it means you're well versed in WAS on z/OS. Pat yourself on the back.

SMP/E product file system

Here's what you'll see in the SMP/E product file system:



That looks like a lot, but if you focus on just the following it makes sense:

- **ola.rar** – the JCA adapter that installs into the WAS z/OS node to provide the inbound and outbound support. It installs like any other JCA resource adapter. The `olaRar.py` Jython script will do this for you, but truth told, doing it by hand is very easy.
- /**olaModules** – these have to be copied out to a pre-allocated module data set or external address spaces such as CICS or batch won't be able to use WOLA. The `olaInstall.sh` shell script (which is in the configuration file system's `/profiles/default/bin` directory)

[13] Intermediate symbolic links are a kind of "alias point" between the configuration file system and the product file system. They help create isolation between nodes. See WP100396 at `ibm.com/support/techdocs` for more on intermediate symlinks.

will copy these modules out to your data set. Or you can do it manually with `OGET`. Once copied out, you may either `STEPLIB` to them, or put them up into LNKLIST.

- **ola_apis.jar** – if you have a Java programmer who wishes to code their EJB to use WOLA, then they'll need this imported into their Rational Application Developer (RAD) environment.
- /**samples** – if you wish to play with the samples (and we recommend you do), then you need to copy those out to a FB 80 data set you pre-allocate. Use `OGET` to pull each file out one at a time. The `OLASample.ear` is the lone exception ... it's a binary file and `OGET` will not be successful in copying it to a FB 80 data set. Just download it to your PC using binary format.

16. What's involved with enabling WOLA in a node?

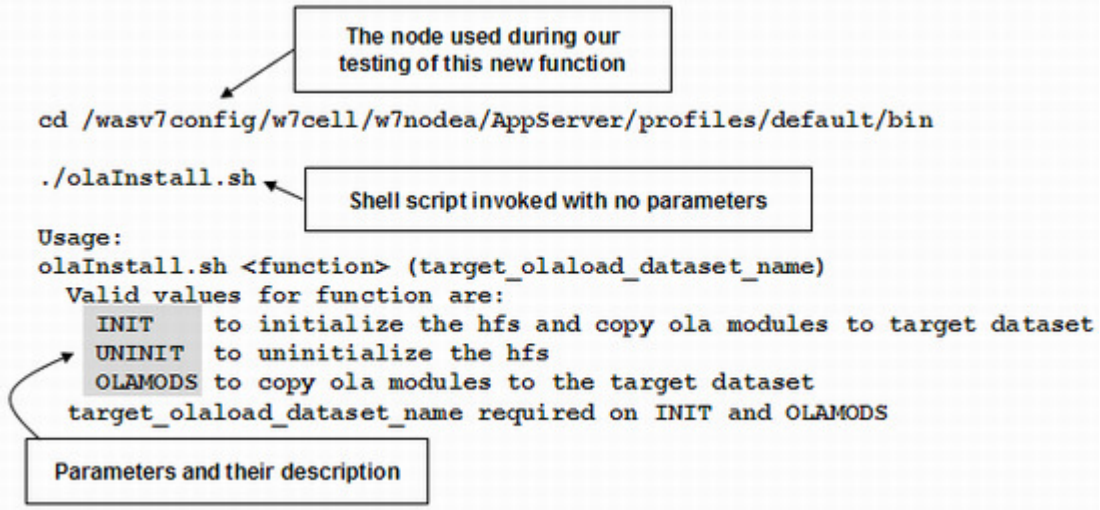
The key piece of this is the `olaInstall.sh` shell script, which we mentioned earlier is under the node's `/profiles/default/bin` directory, creates the symlinks from the node back to the WOLA files in the SMP/E file system.

A secondary element of this is the `olaRar.py` WSADMIN script. That installs the `ola.rar` file and creates a WAS environment variable.

The `olaInstall.sh` shell script

This shell script creates the necessary symbolic links from the node's configuration file system back to the product file system, as well as copying the WOLA API modules out to a pre-allocated PDS.

Determining the syntax for this is easy – invoke with no parameters:



The steps you'd take would be:

- Allocate a load library PDS. Here's a picture of ours so you can get a sense for the size needed:

```
General Data                          Current Allocation
 Management class . . : **None**       Allocated cylinders : 2
 Storage class  . . . : **None**       Allocated extents . : 2
  Volume serial . . . : WAS704         Maximum dir. blocks : NOLIMIT
  Device type . . . . : 3390
 Data class . . . . . : **None**
  Organization  . . . : PO             Current Utilization
  Record format . . . : U               Used pages  . . . . : 203
  Record length . . . : 0               % Utilized  . . . . : 56
  Block size  . . . . : 32760           Number of members . : 24
  1st extent cylinders: 1
  Secondary cylinders : 1
  Data set name type  : LIBRARY

  Creation date . . . : 2009/05/19     Referenced date . . : 2009/05/20
  Expiration date . . : ***None***
```

- Open a Telnet or OMVS session and switch user to the WAS cell's Admin ID.
- Change directories to the node's `/profiles/default/bin` directory
- Issue the command: **`./olaInstall.sh INIT 'your.loadlib.dataset'`**

The result is a node with symlinks built to link out to the product files, as well as the OLA modules copied out to your specified load library. And as mentioned, if the node uses intermediate symlinks, the symlinks created by this utility will use the same intermediate symlinks as the node in general uses.

The other two parameters – `UNINIT` and `OLAMODS` – are used to do what's described in the help that was issued when no parameters are provided.

The `olaRar.py` WSADMIN Jython script

This script does two things:

- It creates a variable with a scope of "cell":

☐	WAS_DAEMON_ONLY_enable_adapter	true

Note: This variable only needs to be created *once* for a cell, but the Jython script will create it every time the script is run. Don't worry ... it doesn't hurt anything.

- It installs the `ola.rar` file into the specified node and creates a connection factory under it:

☐	OptimizedLocalAdapter

The `ola.rar` resource adapter installed into the node

☐	ola	eis/ola	Node=w7nodea	OptimizedLocalAdapter

The connection factory created, with a JNDI name of `eis/ola`

These two things are easily accomplished manually. So if you're not that familiar with WSADMIN, then you may consider going the manual route.

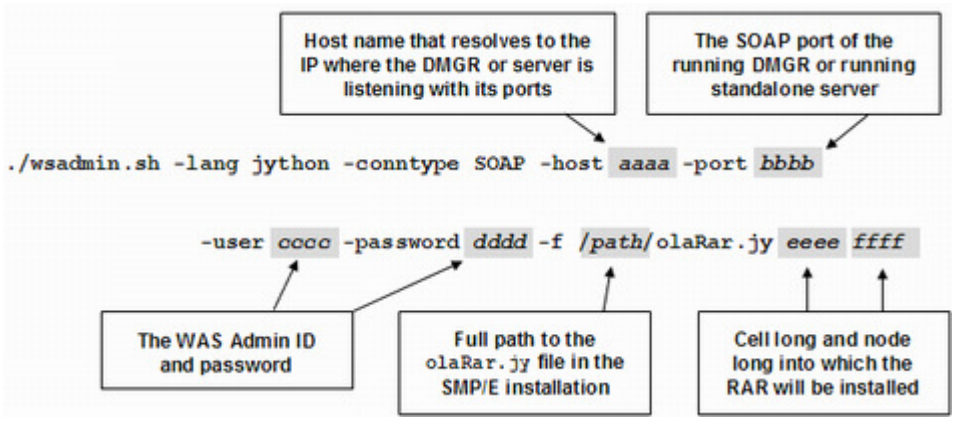
If you want to try the WSADMIN script, then here are the basics of it:

- Make sure the Deployment Manager (or Standalone Server) is up and running
- Open a Telnet or OMVS session and switch user to the WAS cell's Admin ID.

- Change directories to the node's `/profiles/default/bin` directory
- Collect the following information and have it at the ready:

Information	*Why needed*
Cell long name	Parameter passed into Jython script
Node long name	Parameter passed into Jython script
Full path to `olaRar.jy`	Used on –f switch to point to Jython file
WAS Admin ID and Password	Used to gain access into the DMGR or server
DMGR (or Standalone) host name	Used to point to DMGR or server
DMGR (or Standalone) SOAP port	Used to point to DMGR or server's access port

- Issue the following command, *all on one line* (which is why Telnet is typically easier to work with ... OMVS has input length limitations):



Note: People familiar with WSADMIN will know there are variations on this command ... RMI vs. SOAP, ability to avoid userid and password using RMI, coding this in JCL batch, etc. What's shown here is the bare-bones basics ... something that requires little knowledge of WSADMIN to get it to work.

If it ran to completion, you'd have an installed resource adapter, a connection factory with a JNDI name of `eis/ola`, and a WAS variable `WAS_DAEMON_ONLY_enable_adapter` created at the cell scope.

At this point you'll need to insure synchronization to the node has occurred, then stop and restart everything in the cell.

Note: That insures every Daemon in the cell picks up the `enable_adapter` variable. In truth only the Daemon that supports the server you'll validate with needs to be restarted at this point. Other Daemons in the cell may wait. But don't forget to refresh other Daemons before testing WOLA with them or you'll see errors attempting to register into the Daemon.

Validating it using the samples comes next.

17. What's the simplest way to validate the enablement of WOLA?

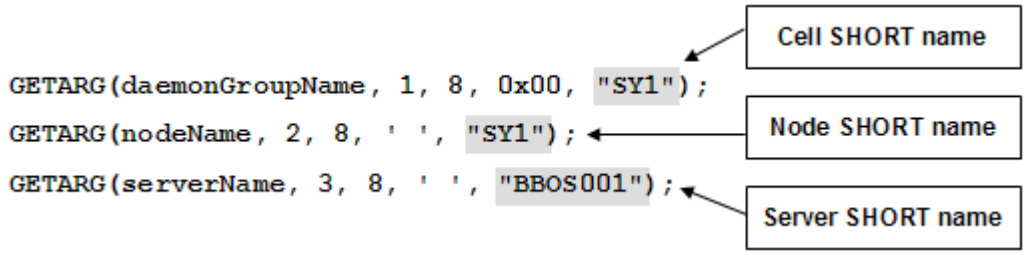
Without question, the easiest validation is the `OLACC01` sample program. As mentioned earlier, that program is a C/C++ batch program that registers, invokes the `OLASample1.ear` file EJB and then unregisters and quits.

It takes a little bit of work to get all the pieces lined up, but it's all relatively straight-forward system programmer tasks.

The *high-level* outline of what to do to make this sample work is the following:

- Allocate a FB 80 data set that will receive the sample members you copy out from the SMP/E product file system.
- Copy *all the sample files* (except `OLASample01.ear`) from the SMP/E product file system to your allocated FB 80 samples data set:

  `OGET /<mount>/mso/OLA/samples/name.jclsamp 'hlq.SAMPLES(name)'`
- Download the `OLASample1.ear` file in binary format to your workstation. It's located in the same directory as the other samples.
- Using the WAS Administrative Console, install the `OLASample1.ear` file into a server in the node you've WOLA-enabled. Take all the defaults.
- Allocate a samples module library that'll accept your compiled samples.
- Edit the `OLACC01` member and modify it so all the data set references are correct for your system environment.
- Find the following in the `OLACC01` program and modify it to match your WAS environment:



- Compile the `OLACC01` program and make sure it compiles with RC=0.
- Make sure the server is started and the sample EJB application in that server is started.
- Edit `OLABATCH` so it points to and runs your compiled copy of `OLACC01`.
- Submit `OLABATCH`
- Check the output for `OLABATCH`. You should see something like this:

```
Invoking service "ejb/com/ibm/ola/olasample1_echoHome"
Invoke response length matches expected: 61
Invoke response data matches expected
```

If that works as expected, it validates several things: a) the node is properly enabled for WOLA; b) the `ola.rar` is installed and operational; c) the `OLACC01` program was able to register and invoke the sample EJB; and d) the sample EJB was there and responded back.

18. What other samples are provided?

Here's the contents of the `@@README` member, which explains what each sample file is for:

```
*** Index of files from <install_root>/mso/OLA/samples:

BBOAAPI  - C header file needed for compiling C/C++ samples that
           use OLA APIs.
CSDUPDAT - CICS DFHCSDUP utility job that defines all the resource
           definitions needed for OLA under CICS.
DFHPLTOL - JCL/source for assembling a sample PLT with the OLA
           enable TRUE exit program, BBOACPLT and the OLA
           BBOC command processor PLT, BBOACPL2.
```

```
OLABATCH - JCL to run one of the samples in batch. Ensure this runs
           on the same LPAR that WAS z/OS is running.
OLACB01  - JCL/source for CICS link to sample Cobol program that uses
           a commarea. This is a sample target program when using the
           OLA CICS Link Server. It echoes back the sent message.
OLACB02  - JCL/source for CICS link to sample Cobol program that uses
           a container. This is a sample target program when using the
           OLA CICS Link Server. It echoes back the sent message.
OLACB03  - JCL/source for CICS sample Cobol program that demonstrates
           how to make a CICS task into an OLA server using the Host
           Service API.
OLACB04  - JCL/source for CICS sample Cobol program that demonstrates
           how to make a CICS task into an OLA server using the Receive
           Request and Get Data APIs
OLACB05  - JCL/source for CICS sample Cobol program that demonstrates
           how to use the APIs to Register, Get a Connection, call an
           EJB with Send Request, Get the response with Get Data, release
           the connection with Connection Release and Unregister.
OLACB06  - JCL/source for CICS sample Cobol program that demonstrates
           how to use the APIs to Register, call an EJB with Invoke and
           Unregister.
OLACC01  - JCL/source for C program that does Register/Invoke/Unregister.
           This can be run under batch/USS/CICS.
```

Note: OLACC01 is the sample we just discussed as being the easiest to set up and use to validate the basics of your environment.

```
OLACC02  - JCL/source for C program that does Host Service/Send Request/
           Send Response/Get Data API calls. This program essentially
           invoke itself, calling in to an EJB that then calls back to
           this program. This can be run under batch/USS/CICS.
```

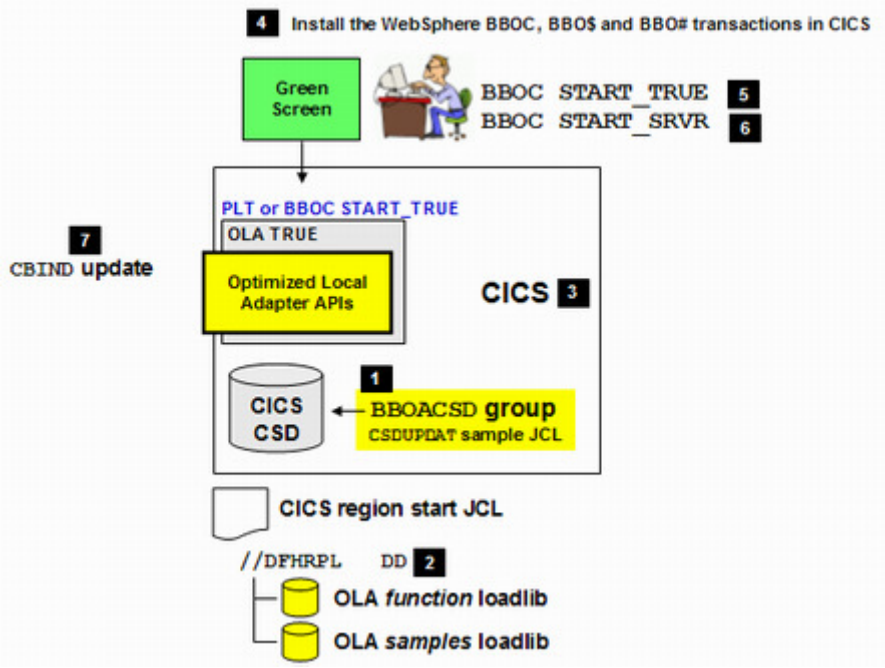
Note: OLACC02 is the second simplest sample because it makes use of the same EJB as OLACC01. All you need to do is modify the OLACC02 sample, compile it, and submit it with OLABATCH.

```
OLAMAP   - JCL/source for a CICS BMS screen map definition. This is a
           3270 test driving screen for passing requests to and from
           WAS from CICS.
OLAUTIL  - JCL/source for Cobol CICS test application utility. Able to
           test Register, Invoke, Host Service, Send Response APIs from
           this panel and update the daemon group/server/node names as
           well as the service name to run with.  This utility can be
           used for testing calling in both directions. Code from here
           can be used as samples for using these APIs.

*** Note the OLAUTIL sample does GET CONTAINER INTOCCSID(37) from        ***
*** CCSID(1208)- UTF-8. Before running this sample to demonstrate a      ***
*** WAS to CICS flow, ensure you are running z/OS Unicode services and   ***
*** can support this conversion. Otherwise an AEZW CICS abend may occur. ***

*** For samples calling from WAS to CICS, you will need the file :       ***
***     <install_root>/mso/OLA/samples/OLASample1.ear and      ***
*** bring up the following URL on your browser:                          ***
***         http://nn.nn.nn.nn:nnnn/OLA_Sample1_Web/                    ***
```

19. What's involved with enabling CICS to use WOLA?

Here again, the InfoCenter has an excellent write-up on the process of enabling CICS to use WOLA. The keyword to search for is tdat_enableconnectorcics.html.

The process involves a handful of steps that should be fairly common to any CICS systems programmer. Here's a high-level picture:



Here's a brief description of what each is referring to:

1 Use the supplied `CSDUPDAT` sample JCL to update the CICS CSD with the OLA definitions.

2 Update the CICS JCL's `DFHRPL DD` to point to the WOLA load library and the WOLA samples library. The WOLA modules are copied into a pre-allocated data set by the `olaInstall.sh` shell script. The sample modules are copied out manually. The key is that for CICS to be able to use the modules, it needs access to them. Updating this `DFHRPL DD` does that.

3 Recycle CICS to pick up the changes.

4 Install the `BBOC`, `BBO$` and `BBO#` transactions. The `CSDUPDAT` job includes them in a group – `BBOACSD`. While you're doing that, install the group `BBOASAMP`, (the samples) which is also part of the `CSDUPDAT` job.

5 Start the TRUE using the `BBOC START_TRUE` command. The sample `DFHPLTOL` is provided to enable the TRUE at CICS startup if you wish.

6 If you'd like to start the WOLA link server, you can do that with the `BBOC START_SRVR` command.

7 Finally, to access the WAS region you'll need to make sure the ID for the CICS region has `READ` access to the `CBIND` class profiles for the WAS cell. Take a look at the `BBO*BRAC` member in the customized job `DATA` data set. That'll give you a sense for the `CBIND` profiles that need updating.

20. What about samples that run in CICS?

There's quite a few. Back under "What other samples are provided?" on page 23 you'll see a listing of all the samples. All of those can run in CICS. The ones that start `OLACB*` are all written in COBOL; the ones that start `OLACC*` are written in C.

For your initial CICS testing, what we'll suggest is that you focus on `OLAUTIL` and `OLAMAP`. That's a 3270 BMS map and a terminal application that can be used to drive a request over to the EJB sample installed in WAS.

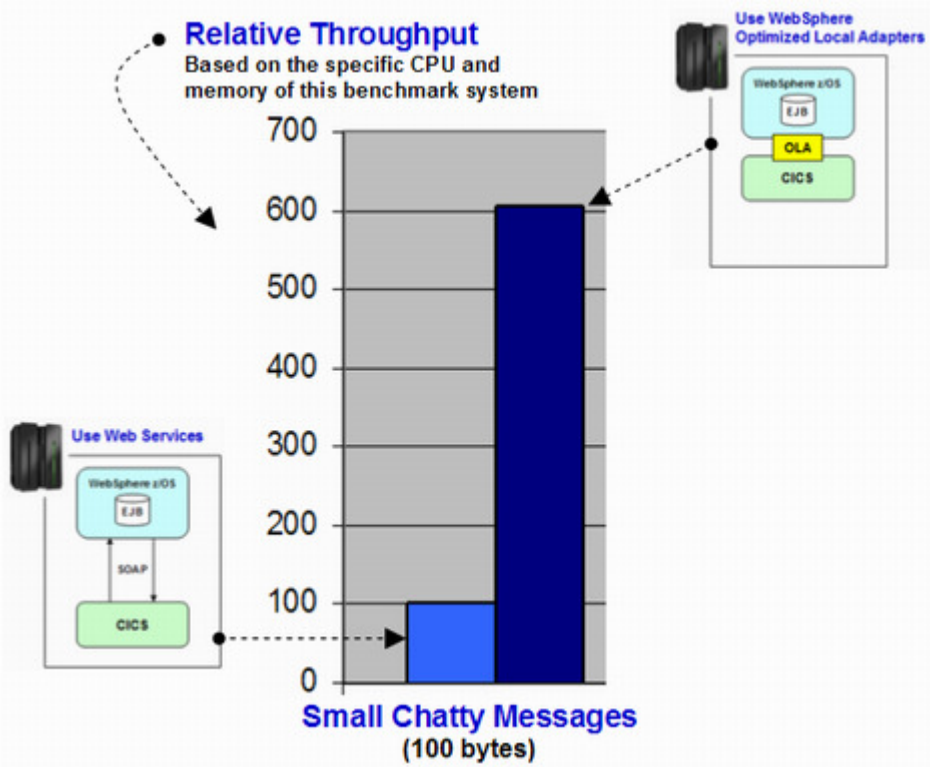
So if you've successfully run `OLACC01` (the simplest verification), then you have the sample EJB in place and you can then use `OLAUTIL` to drive it from CICS. You have to do a little CICS work compiling the screen map and the `OLAUTIL` program, but that's all common CICS system programmer work. It's well documented in the InfoCenter.

Once done, `OLAUTIL` is a very nice way to verify your CICS setup.

21. Do you have performance numbers that you can share?

Some performance testing has been conducted and other testing is underway. As we get the numbers to a point where they can be shared, we will share them.

But we can give you a little something now. The following is a chart[14] that represents the relative throughput difference between web services and WOLA when CICS is the "consumer" of a service and WAS is the "provider" of the service:

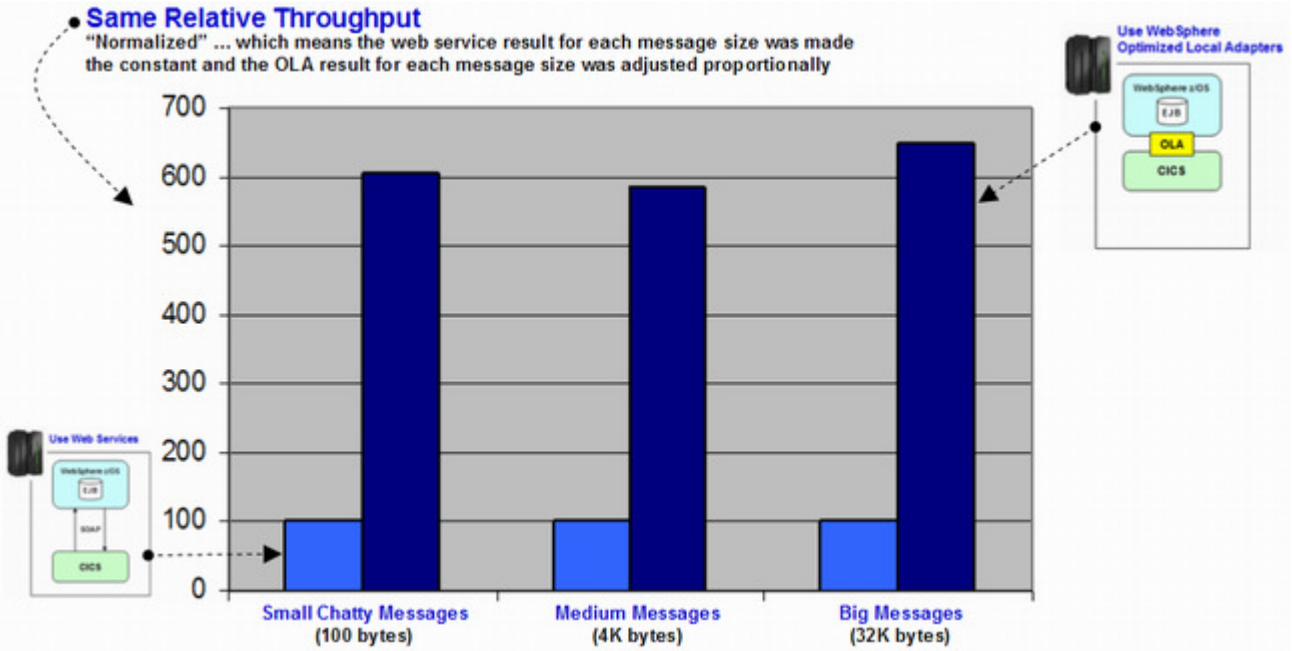


Performance charts *always* require a few notes to explain things, so here they are:

- This is *relative*, not actual throughput. The actual throughput is a function of the server's make and model, and the CPU and memory available. For this test the web service and WOLA tests were conducted on the same system. The web service value was "normalized" to a value of "100" and the WOLA number adjusted proportionally. The focus is not on the actual number; the focus is on the relative difference.
- Web services is a powerful platform-neutral exchange mechanism. But it is not the most efficient. We would agree that WOLA is not a replacement for all uses of web services. We would go on to say that WOLA may prove to be an efficient replacement for web services *in some cases*.
- Web services within a z/OS LPAR? Is that realistic? Some are architecting solutions that do just that. The concepts of SOA are assumed by some to mean "web services everywhere" and the passing of SOAP messages within an LPAR is the result.
- 100 byte chatty messages? Is that typical? In many ways, yes. Web services often result in lots of small message exchanges.
- Is this workload realistic? Not really. This was a trivial message echo application. It provides a relative comparison for this simple workload, but it doesn't attempt to mirror a real-life workload. Other benchmark tests are underway to better approximate that.

[14] Performance numbers offered here were the result of a specific controlled test and do not necessarily indicate the results you may see in your environment. Performance is a function of many factors. Results vary. The findings offered here should not be viewed as a performance guarantee, implied or expressed.

What about larger messages? Would the larger payloads show the advantage of WOLA decreasing? It turns out, the answer is "no:"



We see WOLA outperforming local web services across a range of message sizes.

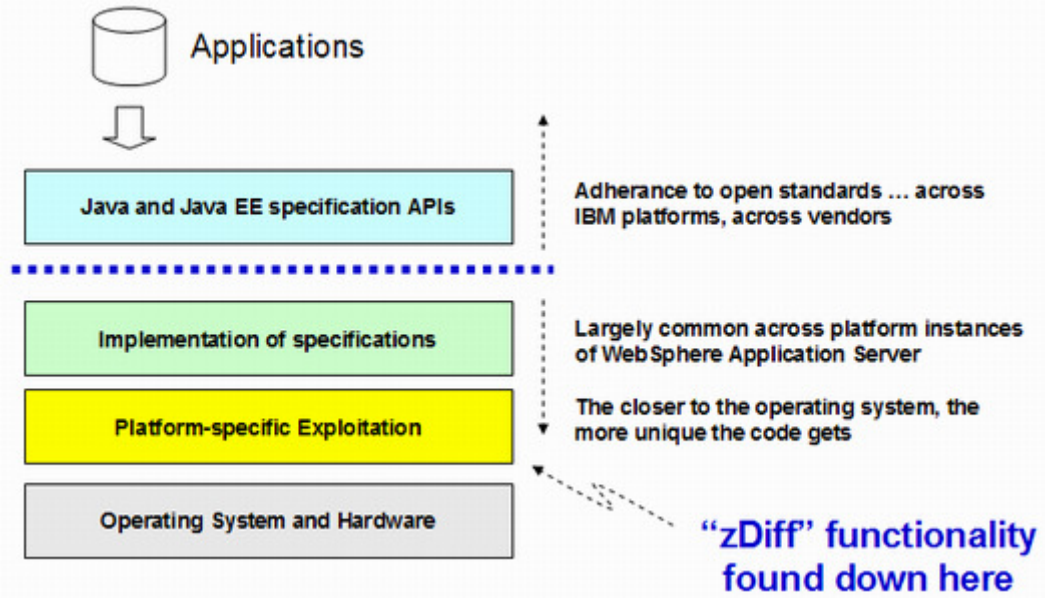
Key Messages:	WOLA is a highly efficient cross-memory data exchange mechanism. These charts illustrate how much better WOLA is for data exchange than web services, *all else equal*. As we have said, WOLA is not designed to be a replacement for web services. What it provides is a technology for optimized data exchange into and out of a WebSphere Application Server for z/OS runtime environment. We encourage a review of cases within your enterprise architecture where co-location and use of WOLA may benefit your business and your customers. That is the intent of WOLA – to be another technology for the advancement of your business objectives. More performance results will be published as they become available.

Conclusion

The "WebSphere Optimized Local Adapters" – informally known as "WOLA" – is yet another piece of technology that augments the *co-location value* of WebSphere Application Server for z/OS[15].

Further, it shows how the *efficiencies of the System z and z/OS platform* can be *exploited* by WAS *while maintaining the open standard specifications at the application layer*.

That's a key point – "WebSphere is WebSphere" at the open standard specification layer and above, but how things are implemented *below* that line is platform specific. On System z and z/OS the goal is to exploit the strengths of the platform as much as possible.



WAS for z/OS V7 introduced a set of functions known as "z Differentiators," or "zDiff" for short. The picture above illustrates how the zDiff items were implemented *below the open standard line*. That's what allows "WAS to be WAS" across all the platforms at the application layer, while letting WAS on z/OS exploit the platform below:

The "zDiff" items introduced in WAS for z/OS V7 included:

- **SMF 120 Subtype 9** – a new SMF record that addressed the shortcomings of the previous WebSphere 120 records while introducing very low overhead to using the new records.
- **Thread Hang Recovery** – a mechanism by which JVM threads marked as "hung" can have WAS z/OS try to shake them loose. Failing that, WAS for z/OS can delay the recycling of the servant region based on other custom settings.
- **FRCA exploitation by the controller region** – FRCA (Fast Response Caching Accelerator) is a function of TCP on z/OS which has proven to be extremely efficient and extremely scalable. This zDiff item allows FRCA to become an external cache group of WebSphere's Dynamic Caching facility.
- **DCS/XCF** – DCS (Distributed Consistency Services) is an awareness function of WebSphere across all the platforms. It's what allows the elements of a WAS cell to understand what's up and available, without relying on a single process to serve as the

[15] See WP101476 at `ibm.com/support/techdocs` for more on the co-location value of WebSphere z/OS.

coordinator of that information exchange. For WAS z/OS, the ability to move DCS signaling from TCP to the "Cross Coupling Facility" (XCF) mechanism of Parallel Sysplex means that less overhead is incurred doing DCS signaling over a large topology cell.

With V7.0.0.4 and the introduction of the WebSphere Optimized Local Adapters, we can add another bullet to the list:

- **Optimized Local Adapters** – efficient cross-memory bi-directional communications between WebSphere Application Server for z/OS and external address spaces.

Open standard consistency across WAS platforms is maintained because the WOLA function is implemented as a standard JCA resource adapter, employing the standard JCA interface specifications.

Let's wrap up this paper ...

Key Messages:

- WebSphere Application Server is IBM's flagship Java 2 runtime environment.
- System z and z/OS are IBM's flagship large system platform and operating system.
- The marriage of WebSphere Application Server and System z and z/OS provides exploitation of the qualities of service of the platform while using open standard solutions. This is the value of "co-location."
- WOLA is another in a growing list of "platform exploitation" technologies for WAS ... while maintaining the common specifications for WAS across all the platforms.

Document Change History

Check the date in the footer of the document for the version of the document.

May 30, 2009	Original document.
June 1, 2009	Added WP101490 Techdoc number and republished.
June 2, 2009	Miscellaneous corrections and updates.
June 4, 2009	Added a numbered question for basic WAS z/OS requirements. Key purpose was to call out the need for the servers using WOLA to be operating in 64-bit mode. Also added a gentle reminder that this document is a "brief introduction" -- not a complete reference – to WOLA. Please make use of the InfoCenter for details.
July 02, 2009	Miscellaneous corrections and updates: a question on using WOLA with Java batch, and the drawing out of some of the finer points of differentiation between WOLA and CICS Transaction Gateway. Also added a note about when the DMGR node might require enablement for WOLA. Finally, added a performance chart comparing WOLA and web services.

End of WP101490