Hidden Gems 2: More Great But Little-Known Features of WebSphere Application Server on z/OS

This document can be found on the web at: www.ibm.com/support/techdocs
Search for document number WP101464 under the category of "White Papers"

Version Date: April 24, 2009
See "Document change history" on page 17 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com

Edited by Wayne O'Brien
IBM Poughkeepsie
Introduction .............................................................................................................................................3
WebSphere Version 6.1—Making controller ASIDs reusable .................................................................3
  Enabling the controller ASID for reuse ...............................................................................................4
  Using the script to save time and avoid potential issues .................................................................4
  Using a release prior to z/OS V1R9 .................................................................................................5
WebSphere Version 6.1—Connecting servants to WLM early ............................................................5
  Solution to the problem .....................................................................................................................5
  Possible side effects ............................................................................................................................6
WebSphere Version 6.1—Using Getting Started Subcapacity Pricing .................................................6
WebSphere Version 7.0—Starting servants faster ................................................................................7
  Starting multiple servants in parallel .................................................................................................7
WebSphere Version 7.0—Setting the custom dispatch thread count ....................................................8
WebSphere Version 7.0—Setting minimum and maximum numbers for servants ...............................9
WebSphere Version 7.0—Report SHRLIBRGNSIZE at startup ............................................................10
WebSphere Version 7.0—Problem determination documentation from the JVM ...............................10
WebSphere Version 7.0—Improved documentation from message BBOO0269E ................................11
WebSphere Version 7.0—Better identification of the source of error log entries ...............................12
WebSphere Version 7.0—Modify timeout delay ..................................................................................12
WebSphere Version 7.0—A Default Classification for Internal Work ..................................................13
WebSphere Version 7.0—Segmenting Output Based on Volume .........................................................14
WebSphere Version 7.0—Where Do My RUNOPTS go? ....................................................................14
WebSphere Version 7.0—Improved wild-carding in WLM classification XML .................................15
Gems (un)covered elsewhere ..............................................................................................................15
Document change history ..................................................................................................................17

Many thanks go to ...Rob Alderman, Don Bagwell, Mike Ginnick, Barry Gower, John Hutchinson, Mike Loos, Colette Manoni, and Tim Spewak.
Introduction

During the development of WebSphere Application Server for z/OS, we sometimes add a few useful little functions to the product without much fanfare or publicity. An earlier white paper, “Hidden Gems: Little Known Features of WebSphere Application Server on z/OS” (WP101138) revealed several of these “hidden gems” in Version 6.1.

This paper begins by looking at some recent updates to Version 6.1 and then dives into WebSphere Application Server for z/OS Version 7 to unearth some new unseen gems in this latest release of the product.

In Version 6.1, we added gems in the following areas:

• Making controller ASIDs reusable
• Allowing servants to connect to WLM early
• SMF record changes for IBM Getting Started Sub-Capacity Pricing (GSSP)

In Version 7.0, we added gems in the following areas:

• Starting multiple servants in parallel, and setting minimum and maximum numbers for servants.
• Controlling the number of dispatch threads in a servant
• Improved tools and diagnostics for troubleshooting.
• Ability to dynamically update the dispatch timeout delay
• Spinning server output by volume instead of by time
• Improved WLM classification wildcarding

If you are a System Programmer or WebSphere System Administrator, this paper has some “gems” for you.

Happy prospecting!

WebSphere Version 6.1— Making controller ASIDs reusable

In the z/OS operating system, each process, or address space, is uniquely identified by a token called an address space identifier or ASID. Whenever your system is initialized (IPLed), z/OS starts with a large, but finite number of ASIDs to use for identifying each new process that runs on the system. Whenever a process ends, z/OS can usually reclaim the ASID for re-use with a subsequent process.

In some cases, however, an ASID might not be eligible for re-use. Instead, the ASID remains unavailable until the next IPL of the system. Whenever this happens, the system's reserve of available ASIDs shrinks by one.

z/OS issues the following message whenever an ASID becomes non-reusable:

IEF352I ADDRESS SPACE UNAVAILABLE

Were this process to continue indefinitely, the system would eventually exhaust its supply of unused ASIDs and would then be unable to start new work. In this situation, your only recourse is to relIPL the system, which renews the system's stock of available ASIDs so that new work can enter the system.

In earlier releases of z/OS, ending the WebSphere controller always left behind a non-reusable ASID. That is because the controller is a target for space-switching (cross-memory) PC routines. When the controller ends, z/OS cannot be sure that no other in-progress threads are space-
switched across the controller address space. Thus, for integrity reasons, z/OS marks the ending controller’s ASID as non-reusable.

In z/OS V1R9, we made some “bookkeeping” changes to allow the system to reuse controller ASIDs. Doing so requires that you to set some switches first. Otherwise, the controller’s ASID remains non-reusable by default.

**Enabling the controller ASID for reuse**

Allowing z/OS to reuse formerly non-reusable ASIDs on your system requires you to do two things on your z/OS system:

1. Enable the function globally by specifying the REUSASID=YES keyword in your active DIAGxx parmlib member. For information about DIAGxx, see z/OS MVS Initialization and Tuning Reference, SA22-7592.
2. Indicate which non-reusable ASIDs are eligible for re-use. The step is necessary because not all applications are meant to run this way. In fact, some applications will abend with the function turned on. WebSphere itself does not have this problem. However, if you are running on z/OS 1.9 you should make sure you have WLM APAR OA28528 applied. Additionally, code which runs in the WebSphere address space (such as in MVS exits) has to comply with the ASID Reuse rules or else a problem similar to the one described in OA28528 might abend the address space.

To have z/OS reuse an ASID, specify the keyword REUSASID=YES on the MVS START command. For example, if you are accustomed to starting the XYZ address space like this:

```
START XYZ
```

You would now start it like this:

```
START XYZ,REUSASID=YES
```

REUSASID=YES is supported on z/OS V1R9 or later.

To enable the WebSphere controller ASID for reuse, specify the REUSASID keyword when you start your WebSphere servers on z/OS. You can do so manually or through your system’s automation program.

**Using the script to save time and avoid potential issues**

Before enabling controller ASIDs for reuse, you should be aware of some potential issues:

- On initialization, a controller searches for an associated daemon process. If a daemon is not started, the controller starts a daemon through the MVS START command, using parameters from the configuration. Because the configuration does not include the REUSASID=YES keyword, the daemon ASID is not reusable in this situation.
- As an alternative to the MVS START command, WebSphere administration services allow you to start or restart servers automatically (WebSphere issues the MVS START command internally). The parameters for starting servers this way are found in the configuration, which (once again) does not include the REUSASID=YES keyword.

These issues are resolved in WebSphere V6.1 service level 6.1.0.19, which includes APAR PK69207. This APAR does two things:

- Adds the REUSASID=YES keyword to the internal MVS START command for the daemon. This change has no affect until you update the DIAGxx member, as mentioned earlier.
Includes a shell script called `updateZOSStartArgs`. Run this script to update the internal MVS START command for one or more of your servers.

Why did we provide this capability to you in a script? That is, instead of just changing the internal MVS START commands ourselves? Most z/OS installations prefer to evaluate changes to the configuration before making them, so providing a script for you to use yourself seemed like the least intrusive approach.

To read more about the script and its options, see this link in the WebSphere information center:


### Using a release prior to z/OS V1R9

If your installation is running a down-level release of z/OS, specifying the REUSASID keyword does not cause problems. For z/OS V1R7 and z/OS V1R8, the command parser detects the keyword and ignores it. For releases prior to z/OS 1.7, WebSphere detects the keyword and removes it before issuing the MVS START command.

### WebSphere Version 6.1—Connecting servants to WLM early

Some Web applications get busy right away, even before the associated servant has completely initialized. A common example is that of an application servlet with an init method that launches as soon as the application starts. Not to worry, the application server can handle this sort of thing.

But now suppose that the servlet init method creates an asynchronous bean, which is a Java object or an enterprise bean that runs asynchronously in the server. This programming practice (creating an asynchronous bean) is generally recommended over starting your own Java thread inside the server.

On z/OS, the thread in which an asynchronous bean is dispatched usually joins a WLM enclave, which allows z/OS to manage the work. The problem with application init methods and asynchronous beans is that applications start— and init methods run— before the servant (which is still initializing) can connect to WLM and create an enclave.

If a servlet runs an init method which launches an asynchronous bean, z/OS needs to associate the bean with a WLM enclave. Otherwise, the bean, lacking an enclave, runs at the priority of the address space.

What to do?

**Solution to the problem**

WebSphere service level 6.1.0.19 includes APAR PK67163, which adds a new configuration variable called `server_region_connect_to_wlm_early`. This variable allows you to specify whether the servant should connect to WLM at the beginning or the end of servant initialization.

To have the servant connect to WLM at the beginning of servant initialization, set the variable to 1. After the servant connects to WLM, any asynchronous work that starts during the remainder of the servant initialization process will be associated with an enclave.

By default, the variable is 0, which prevents any asynchronous work that starts during servant initialization from being associated with a WLM enclave.
Possible side effects

Allowing the servant to connect to WLM early could have some unwanted side effects. If you use multiple servants, understand that the servants will all start almost concurrently, which can use more processor or memory than you might want.

Suppose, for example, that you have configured WebSphere to tell WLM to ensure that a minimum of three servants exist for the server. On initialization, the first servant connects to WLM and indicates that three servants are required. (A servant must perform this communication with WLM because the API for doing so is available only to servants.)

If you allow your servants to connect to WLM early, the first servant barely gets going before it connects to WLM, which causes WLM to start the second servant while the first servant is still initializing. The second servant will also almost immediately connect to WLM and the third servant is started.

The new variable solves the problem that we described earlier and, with a single servant, it will have no other consequences. With multiple servants, however, the change in the interactions with WLM will cause all of your servants to initialize in parallel.

If you want a set of servants to start faster and you have the available processor and memory to support it, you might find variable server_region_connect_to_wlm_early to be helpful. On the other hand, if you are already constrained for processor or memory resources when servants are starting, having a bunch of servants start concurrently is the last thing you want.

To learn more about variable server_region_connect_to_wlm_early, see this link in the WebSphere Information Center:


WebSphere Version 6.1—Using Getting Started Subcapacity Pricing

Have you heard about the new pricing option called IBM Getting Started Sub-Capacity Pricing (GSSP)? You can read the announcement letter here:


GSSP requires WebSphere Application Server level 6.1.0.16.

With GSSP comes a change in WebSphere. Starting with WebSphere V6.1 level 6.1.0.16, z/OS no longer creates SMF Type 89 Subtype 2 records for WebSphere. Instead, it creates SMF Type 89 Subtype 1 records.

Previously WebSphere used the z/OS register service (IFAEDREG) to register itself with a z/OS image. This action causes SMF type 89 subtype 2 records to be created, which are then used by the Sub-Capacity Reporting Tool (SCRT) to create reports. IBM uses the SCRT reports as part of the normal Sub-Capacity Pricing program.

However, GSSP is based on CPU usage, which is not included in the subtype 2 records. Thus, to support the new pricing option, we changed WebSphere to use a different z/OS service, IFAUSAGE, which causes SMF 89 subtype 1 records to be written, which include details about CPU usage.

If you were using the subtype 2 records for some other purpose besides the SCRT reports, you can get them back by specifying the configuration variable register_ifaedreg_also=1. Doing so causes WebSphere to call both IFAEDREG and IFAUSAGE, which causes z/OS to create both SMF Type 89 Subtype 2 records and SMF Type 89 Subtype 1 records. The SCRT tool will not care.
If you set `register_ifaedreg_also`, the subtype 2 records are written for GSSP-enabled products, too.

To turn off the creation of SMF Type 89 Subtype 2 records, set this variable to 0.

To read more about the variable `register_ifaedreg_also`, see this link in the WebSphere Information Center:


---

**WebSphere Version 7.0—Starting servants faster**

When you start the application server, the first thing to initialize is the controller. The controller connects to WLM, but WLM does not start a servant until it needs one. And, at this point, it does not, because the queue is empty of work requests and the initial minimum and maximum number of servants is zero.

To get things moving, WebSphere places a dummy piece of work on the WLM queue. This action causes WLM to want a servant when it has none, so WLM starts up a servant. When the servant finishes initializing, the servant connects to WLM. Now WLM wants one servant and it has one servant. WLM is happy.

But suppose that when the servant connects to WLM, it tells WLM that you have configured the server to have a minimum of three servants. Now WLM wants three servants, but it has only one. WLM starts a second servant to solve its problem. After the second servant initializes and connects, WLM sees that it has two servants, but it wants three. WLM then starts a third servant. When the third servant is initialized and connected, WLM is happy again because it wants three servants and it has three servants.

The problem is that all of this processing takes time. The listeners in the controller usually do not open until the minimum number of servants is initialized. Starting the servants one by one means that a relatively long time can pass between the start of the controller and a server accepting work requests.

In this situation, it would be preferable to start the servants in parallel. Previously, in “Connecting servants to WLM early,” we described an option for causing the servant to register with WLM earlier. Though this option has the effect of getting servants started sooner, it is accomplished by “lying” to WLM. The servant claims it is ready for work before it is really initialized.

Let’s look at an alternative approach.

**Starting multiple servants in parallel**

Version 7 adds a new variable called `wlm_servant_start_parallel`, which does just that: Starts servants in parallel. Set this variable to ‘1’ if you want WebSphere to let WLM know that it can be “hasty” in starting the minimum number of servants.

With this setting, when the first servant is initialized, WebSphere tells WLM the minimum number of servants specified in the configuration and tells WLM that it is ok to start them in parallel. In our example, WLM detects that it has one servant, but needs three. Rather than just starting one and seeing how that goes, WLM will do the math and realize that starting two servants at the same time might help it get the number of servants it wants. So the second and third servants will start in parallel.

Be sure that you have enough processor and memory to support this. Otherwise, you might find it takes longer to start two servants in parallel than in sequence. Also, some file system contention is
possible when servants start in parallel. Using a zFS file system instead of HFS helps to reduce the contention.

The variable `wlm_servant_start_parallel` also affects how WLM restarts servants if they are terminated by an abend. Here, suppose that you have three servants up and running. If two of them abend, WLM realizes that it has one servant, but wants three. If you do not set variable `wlm_servant_start_parallel` to ‘1’, the replacement servants will start one at a time as before. With the new variable set to ‘1’, WLM will start both replacement servants in parallel.

If you have the processor capacity and memory to support this behavior, the variable `wlm_servant_start_parallel` gets the minimum set of servants back up more quickly.

The number of servants that WLM will start in parallel for a particular server is limited. If you define the minimum number of servants as a sufficiently large value, WLM will start them in groups. The exact number that will start as a group is internal to WLM.

To find out more about the variable `wlm_servant_start_parallel`, see this link in the WebSphere Information Center:


---

**WebSphere Version 7.0—Setting the custom dispatch thread count**

You can control the number of dispatch threads in a servant by setting the variable `server_region_workload_profile`.

In previous releases, you were limited to one of the following options:

- **ISOLATE** Server starts a single dispatch thread per servant.
- **CPUBOUND** Server calculates the number of threads based on the number of active CPs.
- **IOBOUND** Server does a different calculation, based on the number of active CPs.
- **LONGWAIT** Forty dispatch threads are started.

We had hoped that eventually **IOBOUND** and **CPUBOUND** would have a smarter algorithm to determine a proper number of threads, perhaps based on available system capacity. Unfortunately, it didn't happen. The rationale for providing the **LONGWAIT** option was that some applications simply block the thread for a long time waiting for some external event to occur (which is much worse than an iobound request) and so a large number of threads is needed. Forty seemed like a large number.

We’ve gotten numerous comments over the years about these options. In Version 7, WebSphere will let you specify exactly how many threads are needed. You do this through the new (fifth) option for the workload profile: **CUSTOM**. This option tells the server to use the value of the variable `servant_region_custom_thread_count` to determine the number of threads.

The thread count variable defaults to 40, just to be consistent. But you can set any number you like, up to 100 threads. Why is the maximum 100? Because that's how many we have tested.

To find out more about these options, see this link in the WebSphere Information Center:

WebSphere Version 7.0—Setting minimum and maximum numbers for servants

WebSphere allows you to configure a minimum and maximum number of servants for a server. WLM will dynamically adjust the number of servants within the specified range, up or down based on what’s needed to meet the goals for the system. WLM does this for work running in WebSphere and for work elsewhere on the system.

To set the minimum value, consider how many servants you want to start automatically when the server is started and how many you want WLM to keep available. In determining the maximum value, consider how many servants you can support on your system. Also, consider the number of available connectors for applications in WebSphere and elsewhere in the system.

But what if something changes someday and the minimum just is not enough? Or, you reach the configured maximum and need more servants? To change the values, you must update the configuration and recycle the server. But if you are running at peak utilization and decide you need to increase the maximum number of servants; recycling the whole server is probably going to hurt more than just not having enough servants. It would be nice to be able to dynamically change the number of servants without a recycle.

In Version 7, we introduced a new MODIFY command to let you do that. If the server is not configured as single-servant, you can change the current minimum and maximum number of servants. You enter the command as follows:

```
MODIFY server,WLM_MIN_MAX=(minimum, maximum)
```

Specify these values as decimal numbers. Obviously, the minimum must be less than the maximum.

Your changes are in effect until the next time you recycle the server, in which case, the values in the configuration are used instead. To make your changes permanent, you need to update the configuration.

In general, WLM responds quickly to your request. If the minimum number of servants is not already running, WLM starts more. Increasing the maximum value, however, might not have any immediate effect. Further, decreases in values might also not cause an immediate change because of WLM’s opinion as to how many servants it needs. Some situations, such as session data pinned to the servant, might prevent WLM from reducing the number of currently active servants.

Of course, unless you’ve committed your min and max values to memory, you would probably like to have a quick way to see what you are currently configured for. We added a new command to allow you to do that.

```
MODIFY server,DISPLAY,WLM
```

This command displays the values like this:

```
BBOO0343I WLM MIN/MAX SERVANTS: 3/3
```

To find out more the new MODIFY command, see this link in the WebSphere Information Center:

```
```
WebSphere Version 7.0—Report SHRLIBRGNSIZE at startup

The load modules that make up the Java Virtual Machine (JVM) are loaded from the z/OS UNIX System Services (z/OS UNIX) file system, rather than from data sets. In WebSphere Application Server Version 7, the native load modules can also be loaded from the z/OS UNIX file system. Generally, these modules are not loaded into common storage, but rather, are loaded into the private region of the server’s address space. While it is possible to put the modules into LPA, another solution is available. z/OS UNIX offers a shared library region into which you can place any modules to be shared between address spaces.

You can read about this feature in the z/OS and USS documentation. A good summary of all the issues is available in IBM White Paper “SHRLIBRGNSIZE and the Effect on 31 bit JVM Storage Needs” (WP101320). That paper describes how to find out the current setting for the size of the shared library region as well as how that space is being used.

Knowing the configured size of the shared library region at the time the server initialized may also be useful to know. For that reason each region of the application server (controller, servants, and the CRA) issues message BBO00341I during startup. The message looks like this:

```
BBO00341I VARIOUS RESOURCE MONITORING DATA: (64):():():():():():():():()
```

The ‘64’ in parenthesis is the size of the shared library region in megabytes. But what is the rest of this mess? The MVS rules for console messages say that we can not change the text of a message once we have started issuing it. However, we can introduce new values for fill-ins. We thought that perhaps over time we might come up with other values it would be interesting to know as the server starts up. Therefore we placed a fill-in value inside each set of parenthesis. Presently these fill-ins are all empty strings, but someday there might be values there. In the meantime just think of it as the server happily smiling at you.

WebSphere Version 7.0—Problem determination documentation from the JVM

Debugging problems on z/OS often involves obtaining an SVCDUMP of the failing address space and using IPCS to examine it. But Java application developers that are used to other platforms do not know about an SVCDUMP and are used to other output for debugging.

On other systems you can sometimes use the shell command ‘kill’ to send a signal to a process. This is often used to signal the JVM in a WebSphere server process to take a java dump. On z/OS you can send signals to the controller and servant this way, but because of the internal structure of the server this action rarely results in the dump you expected.

You can use the wsadmin dumpThreads command to trigger a javacore as well as a heap dump from all of the servants of the server which receives the command.

Starting in Version 7, you can also use MVS console MODIFY commands to cause the server to collect assorted documentation.

The first command we will look at is: MODIFY server,STACKTRACE. The stacktrace command causes the controller and all the servants to dump the Java stacktraces of all Java threads. The output goes to the job output. This is a view from a Java perspective and will not show any native threads and will not show the native stack of a Java thread that has made a JNI call.

To get more information you can issue MODIFY server,JAVACORE. This command triggers a call to the JVMRAS GenerateJavacore API. The dump will be taken by the JVM in the controller and in all the servants. The contents are written to a file in the file system. A message in SYSOUT will tell you the exact path and file name. A Javacore contains all the stack traces as well as some other basic information about the JVM.

© 2009, IBM Software Group, Application and Integration
Middleware Software, Poughkeepsie, NY
You can also use the MODIFY server,HEAPDUMP command. Like the JAVACORE command, HEAPDUMP triggers a dump into a file in the file system. In the controller and in each servant a call will be made to the JVMRAS GenerateHeapdump API. As with JAVACORE, a message in SYSOUT will tell you the path and file name where the dump is written. Since the heap is dumped, these files can be quite large.

The last new option for Version 7 is the TDUMP. On z/OS an unauthorized program can call the IEATDUMP API to generate a small dump that is somewhat like an SVCDUMP. The JVM provides a JVMRAS API, namely InitiateSystemDump, that causes the JVM to stop any active processing and call IEATDUMP. Of course the JVM can’t stop any processing outside of its control, so non-Java threads might still be changing things while the TDUMP is taken. The MODIFY server,JAVATDUMP command will cause this dump to be taken in all the servants. Note that, unlike the other commands, this command does not gather documentation from the controller.

A TDUMP is written to an MVS data set (much like an SVCDUMP) and should be looked at with IPCS. The name of the data set is controlled by the JVM and is based on an environment variable the JVM defines. There are also other environment variables that can affect how this API behaves. See the JVM documentation for details.

As noted earlier, to read more about the Modify command, see this article in the Information Center:

**WebSphere Version 7.0—Improved documentation from message BBOO0269E**

Sometimes we think we are clever. Consider the case where the server is initializing and WebSphere is binding to a port specified in the configuration. In some situations the bind might fail. The API sets the errno and errnojr values to explain the problem. If the errno value is EADDRINUSE WebSphere concludes that the port is already in use and issues message BBOO0269E to tell you that. If the bind fails for a different reason, yielding a different errno, the code issues message BBOO0161E which tells you the errno and the errnojr. We thought we were very clever to recognize the cause of the problem and issue a message telling you exactly what was wrong.

Except that sometimes the problem is something else. It turns out that the bind can fail for other reasons and still generate that errno. The errnojr value is different depending on the reason. Being clever, we did not tell you the errnojr value because we thought we knew what was wrong. A customer encountered this other situation and had some trouble debugging their configuration problem because they couldn’t get WebSphere to tell them the errnojr value.

In Version 7 we’ve modified the code to simply always issue the BBOO0161E message with the errno and errnojr values, even if we think we know what the problem is and issue message BBOO0269E. That way you get the error codes you need to debug the problem in case we are not as smart as we think we are. All things considered this is a pretty small ‘hidden gem’, but I have included it just to point out that we do listen to complaints, even small ones, and fix them when we can.
WebSphere Version 7.0—Better identification of the source of error log entries

Another requirement we received concerned messages written to the error log. Currently those messages have a header which looks something like this:

2000/06/01 16:01:06.683 01 SYSTEM=SY1 SERVER=BBOASR1A JobName=BBOASR1S ASID=0X0033 PID=0X0100003C TID=0X24F858A0 0X000004 c=2.101003

The requirement pointed out that server names are not necessarily unique and that jobnames certainly aren’t. The ASID isn’t terribly helpful in determining which server wrote the message once the server has ended. So for Version 7 we updated the header to specify the cell, node, and cluster name of the server writing the record. The message looks like this now:

2000/06/01 16:01:06.683 01 SYSTEM=SY1 CELL=SY1 NODE=NODE1 CLUSTER=BBOASR1 SERVER=BBOASR1A JobName=BBOASR1S ASID=0X0033 PID=0X0100003C TID=0X24F858A0 0X000004 c=2.101003

Of course if the error log messages are just going to the SYSOUT DD card in the job output it is pretty obvious which server wrote the message. But if you have the messages going to a logstream and that logstream is shared between multiple servers it could be hard to figure out which server wrote which messages. This change should help.

For more information about what all the fields in the header mean, see the following article in the Information Center:


WebSphere Version 7.0—Modify timeout delay

One of the many configuration variables you can set to control the application server is called control_region_timeout_delay. The value of this variable is used to set a timer in the controller after a dispatched request has timed out. The controller has decided to abend the servant where the request is dispatched to get the request to end. Because other requests might be dispatched in that servant, you can configure a timeout delay to allow them to complete. During this delay the servant will not accept new requests for dispatch.

Working with customers who have enabled this feature, we have learned that it is sometimes hard to anticipate what the right delay value will be. The variable is read at initialization which requires you to recycle the server in order to change the value. If an application gets updated and starts having timeout problems, it would be nice to be able to adjust the delay appropriately until the problem in the application is fixed. Having to recycle the server to do this might create a bigger problem than you already have.

In Version 7 we introduced a new MVS console Modify command to let you change the value of this variable dynamically. Just issue

MODIFY server,TIMEOUT_DELAY=n

In this example the value ‘n’ is the number of seconds the server should use as the delay. This change is only in effect for the life of the server. Recycling the controller will cause it to pick up the delay value from the configuration. However, this command will let you temporarily change the
value as often as you like. If a timeout has already occurred and is already in a delay before
abending the servant, this command will not change the length of that delay.

As noted earlier, to read more about the Modify command, see this article in the Information Center:
ries/ae/rxml_mvsmodify.html

WebSphere Version 7.0—A Default Classification for Internal Work

WebSphere creates a WLM enclave for all requests that get dispatched in a servant. An enclave
has an associated WLM service class and report class. The service class is used by WLM to help
make decisions about assigning resources to the servant to ensure the requests meet the goals
defined in the service class. The report class is used to separate information about the requests in
reports generated by RMF (or other similar products). To determine which service class and report
class to assign to the enclave WLM looks at classification information provided by WebSphere
when the enclave is created.

One piece of information provided is called a transaction class. This is just an eight character name
assigned to the request. WebSphere supports an XML file pointed to by the variable
wlm_classification_file to determine what transaction class to use. The XML file allows you
to specify a different transaction class (and thus indirectly a different service class and report class)
for different applications or even parts of applications.

The XML file also allows you to specify a transaction class to be used when classifying requests
that are internally generated. Sometimes the controller needs to dispatch something in its own
servant. For example, this could be the dispatch of a management bean (MBean). To separate
internal work from application work, you might want to put these requests in their own report class.
To do that you simply specify the ‘internal’ clause in the XML file and provide a transaction class
name that WLM will recognize (based on the rules you provide in the WLM configuration) and
assign service and report classes appropriately.

What if you just want to separate out internal work from real application work? Creating and keeping
track of an XML file just to specify a single transaction class name is overly complex. Therefore, in
Version 7, in response to a customer requirement, we created a new configuration variable named
default_internal_work_transaction_class. If there is not a WLM classification file with a clause for
internal work, WebSphere will use the transaction class name specified by this variable to classify
internal work.

If you’re already using the XML file to classify internal work, this variable is ignored. But if you want
to classify internal work separately, setting this variable is an easy way to do that. If there is no
clause for internal work in the XML file and the variable isn’t set then the transaction class name
provided to WLM is all blanks.

For more information on the contents of the WLM classification XML file, see this article in the
Information Center:
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.zseri
es.doc/info/zseries/ae/rrun_wlm_tclass_dtd.html
WebSphere Version 7.0—Segmenting Output Based on Volume

If you have read the Hidden Gems paper for Version 6.1, you know about the variables `ras_stdout_ff_interval` and `ras_stderr_ff_interval`. These variables control the interval at which WebSphere writes a form-feed to standard out and standard error. The form feeds count as page boundaries and JES2 will spin the output when the number of pages reaches the SEGMENT size defined on the SYSPRINT and SYSOUT DD statements.

This capability allows you to configure WebSphere and the JCL so that the output of these DD statements will be spun by JES2 at regular intervals. For example, if you tell WebSphere to write a form feed every hour and set the SEGMENT size to 24, JES2 spins the output every twenty-four hours. This allows you to remove excess output from the controller or servant, thus, freeing up spool space without having to recycle the server.

However, to use this option, you need to be able to predict how much output will accumulate over a specific amount of time so that you can configure it correctly. As an alternative, you could specify a volume interval instead of a time interval to have JES2 spin the output whenever a certain amount accumulates, rather than after a certain amount of time has passed. You can do that through two new options we added in Version 7:

- `ras_stdout_ff_line_interval`
- `ras_stderr_ff_line_interval`

The value of each variable indicates an approximate number of lines of output. When about that many lines have been written, a form feed is written to create a page break. Therefore, if you wanted to spin the output every 20,000 lines you could configure the line interval to be 1000 and the SEGMENT size to be 20 (or any other combination that multiplies out to 20,000). WebSphere will count the lines written and write a form feed when the count gets to the configured value.

Not everything that gets written to these DD cards comes from WebSphere. That means that the code does not see all the lines written and it can not count what it does not see. For example, if you have VerboseGC turned on for the JVM, that output is written to the SYSOUT DD directly by the JVM. WebSphere will not see those messages and will not count the lines. There are some other special situations where WebSphere has to approximate the number of lines written which can cause the count to be slightly off.


WebSphere Version 7.0—Where Do My RUNOPTS go?

In previous versions of WebSphere, the JCL for the controller and servant directly invoked the WebSphere load module by running BBOCTL or BBOSR. These are LE programs and as such you could specify LE Runtime options in the JCL PARM string. You could place a slash in the string and anything before the slash was parameters to LE and anything after the slash was parameters to WebSphere. The LE runtime got control first and parsed things apart, passing the things after the slash to WebSphere.

In Version 7, all the load modules moved out of data sets and into the file system. Now the JCL executes BPXBATCH (or a flavor thereof) and the PARM string tells BPXBATCH what program to run and the parameters to pass to it. There is not anywhere in that string that you can put a slash and successfully pass parameters to the LE runtime. What do you do if you want to set some LE option?
The place to put them now is in the was.env file pointed to by the STDENV DD statement in the JCL. This file is created automatically from variables in the configuration. To configure Language Environment runtime options, click Environment->WebSphere Variables, select a scope, and click New. Define the variable _CEE_RUNOPTS to the runtime options you want. For example:

_CEE_RUNOPTS=RPTSTG(ON),RPTOPTS(ON)

This is not really a hidden gem, but it is a change you might notice and wonder about. This paper seemed like at least one good place to point out the new way to set LE runtime options.

WebSphere Version 7.0—Improved wild-carding in WLM classification XML

Earlier, we mentioned the XML file that you can use to determine a transaction class name. WebSphere provides this class name to WLM when a request is classified and associated with a service class and report class.

The XML file allows you to specify attributes to compare against the request being classified. For example, you can specify the URI of an HTTP request (the URI is the part of the URL after the host and domain name). Consider this URL, which happens to link to the WebSphere Information Center article on the WLM classification XML file.


In this example, the URI is everything that follows ‘publib.boulder.ibm.com’. If you wanted to specify a transaction class for this URI, you could include an entry in the XML for this full string. However, you probably do not want to be that specific. You might, for example, want to create separate transaction classes based on the product release, so the URI you use in the XML classification file could be just this much of the string:

/infocenter/wasinfo/V7R0/*

The asterisk at the end tells the comparison code to match any URI that starts with this string. In previous releases of WebSphere Application Server for z/OS, you could only place the wildcard at the end of the string. Thus, if you wanted to match just URIs ending in ‘.html’ you had to specify them all individually.

Naturally, this limitation generated a customer request to improve the wild card support. As a result, Version 7 allows you to place a single asterisk anywhere in the string. For example, write a URI specification like this:

/infocenter/wasinfo/V7R0/*.html

This string would match any URI for the V7 Information Center that ended in ‘.html’. Requests for any other type of file in the V7 Information Center would not match.

This improved wild card support allows you to perform different classifications within the same application, based on file type or some other filter criteria. While you probably do not want different service classes based on file type, you might want different report classes. That would let you get separate reports for static pages versus dynamic pages, for example.

See the preceding URL for more information about the WLM Classification XML file.

Gems (un)covered elsewhere

WebSphere Version 7 includes other items that might be considered “gems;” we described them in IBM White Paper WP101374.

The first of these is the Dispatch Progress Monitor (DPM for short). The DPM allows you to capture diagnostic information about an application when an occasional request takes too long. For
example, if most requests for an application take less than five seconds, but a few times a day a request for the same application takes over a minute you might wonder why.

The DPM will allow you to set an interval; in this case ten seconds might be appropriate. If a request takes longer than ten seconds to dispatch, the DPM will capture a callstack for the dispatch thread.

The interval timer is then reset and, if another ten seconds pass and the request is still in dispatch, the diagnostics will be gathered again. This continues until the request completes.

You can also configure the DPM to capture other diagnostics such as a heapdump or even an SVCDUMP. This allows you to get several snapshots of what the application (or the whole servant) was doing when these occasional long running requests happen.

IBM White Paper WP101374 also discussed the “gem” we call CPU timeout. Dispatch timers in the past have just monitored elapsed time for a request. You might be willing to allow a request to have several minutes of elapsed time to execute before considering it ‘timed out’. However you might want to stop an endlessly looping request that is just consuming CPU before it used up several minutes of CPU time. Version 7 allows you to specify a CPU timeout. This causes the server to take action when the application request has consumed too much CPU time, even if it has not used up anywhere near the amount of elapsed time you allow.
## Document change history

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 22, 2009</td>
<td>Initial Version</td>
</tr>
<tr>
<td>April 24, 2009</td>
<td>Added Whitepaper #</td>
</tr>
</tbody>
</table>

End of WP101464