WebSphere Application Server for z/OS

Dispatch Timeout Improvements in WebSphere Application Server for z/OS Version 7

This document can be found on the Web at:
`www.ibm.com/support/techdocs`
Search for document number **`WP101374`** under the category of "White Papers"

Version Date: May 2, 2013
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

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

Edited by Wayne O'Brien
IBM Poughkeepsie

Many thanks go to Robert Alderman, Mike Ginnick, Tim Spewak, Tim Kaczynski, Don Bagwell, and loads of others.

Introduction

WebSphere Application Server for z/OS Version 7 adds new configuration options and facilities for avoiding timeout-related abends. These enhancements include:

- Ability to "nudge" a timed-out request to completion without abending the servant region
- Stalled thread threshold to avoid abending a servant until a specific number of problem requests is reached
- CPU timeout option to prevent a runaway request from consuming excessive resources
- DISPLAY,THREADS command for information about server activity and for identifying long-running requests
- Dispatch Progress Monitor (DPM) to gather documentation about a request that runs longer than expected.

In this IBM White Paper, we look at how you can make the best use of the new options in your environment.

Review of dispatch timeouts in WebSphere

Let us begin by reviewing the concept of a "dispatch timeout" as it applies to WebSphere.

In the WebSphere environment, your application servers cope with a variety of work requests. In normal operations, the servers handle the beans, servlets, and other types of application workloads efficiently and without incident. What happens, however, when processing does *not* go as planned? That is, when applications have bugs, network connections hang, needed resources are slow or unresponsive, and bad things just *happen*?

Fortunately, the architecture of WebSphere Application Server on z/OS is designed to help you deal with these situations. Each application server consists of a controller region and one or more servant regions in which the inbound application requests are dispatched (Figure 1). By separating the server into these distinct processes, WebSphere makes it easier for you to identify a misbehaving application request and, if necessary, end it.

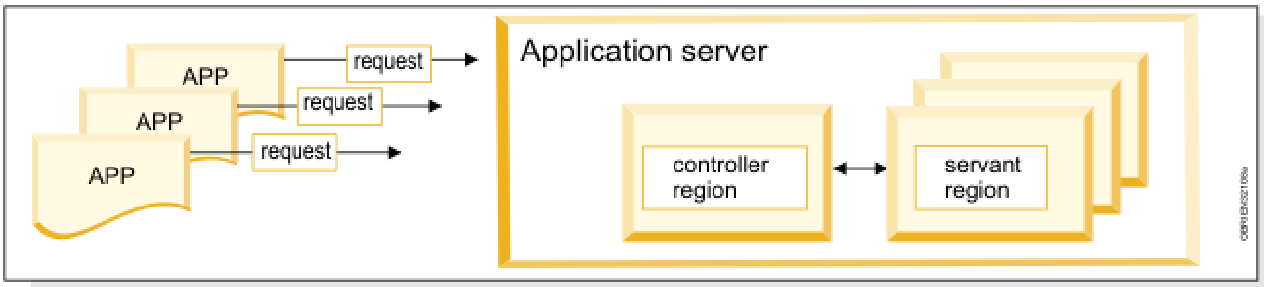

Figure 1. An application server is comprised of a controller region and one or more servant regions

In WebSphere, you can set a timeout value for the time in which a work request must complete. If the request runs beyond this limit, it fails (it "times out"). Timeout processing allows the controller, which is separate from the servant, to take action — for example, to terminate the servant and start a replacement. For added flexibility, WebSphere allows you to set specific timeouts for different types of work requests: HTTP, IIOP, MDB, and so on.

Of course, whenever you have a bug in an application or somewhere else in your network, you should correct the underlying problem. Resolving some problems, however, can be time-consuming, so WebSphere provides some options to help make living with timeouts a little easier. For information about these options, see IBM White Paper "WebSphere Application Server for z/OS V6.1 Configuration Options for Handling Application Dispatch Timeouts," WP101233.

New timeout behavior in Version 7

In previous releases of WebSphere Application Server for z/OS, a dispatch timeout always meant that an abend was in store for the servant region of the dispatched request. Although WebSphere provided some configuration options for delaying the abend or initiating actions that could happen in parallel with the timeout, an abend was always the inevitable result of a timeout.

Now, in WebSphere Version 7, the servant need not always come to an untimely end. To allow the servant to "cheat death," we made some changes in Version 7 to alter the behavior of dispatch timeouts.

One key change is that we now have two timers, not just one. The existing timer (the controller timer) is extended so that it now expires sometime *after* the dispatch time has been exceeded. This extra time can improve the odds that the request will complete dispatch when a second, new timer (the servant timer) "nudges" the request, for example, with an error.

In many cases, the nudging might be enough to *unblock* (interrupt) the request so that it can complete dispatch. To the controller, it appears as though the request finished on time, and there is no need to abend the servant.

Sidebar: Obviously, **I**nterruption **O**bject does not abbreviate to *ODI*, so how did we arrive at that particular acronym?

Given that "IO" is already a well known acronym, calling it that would probably be confusing.

But that's just in English. In French, “Interruption Object” translates to *Objet D’Interuption*. In Spanish, it is *Objeto De Interuption*. In Italian, it is *Oggetto Di Interuption*. All of which abbreviate to *ODI*.

Fortunately, no other computer-related definitions of ODI appear to exist (so far).

And no, ODI bears no relation to a certain dog from a certain comic strip starring a rather overweight cat. His name is spelled with an ‘e’.

☺

Using an ODI to "nudge" a request

When an application initiates an operation that might *block* (become delayed), thus causing its request to remain in dispatch long enough to time out, the runtime can register something called an Interruption Object (ODI for short; see Sidebar). Later, when the risk of a block has passed, the ODI is deregistered.

When a timer expires for the request, the timer processing calls the most recently registered ODI, which attempts to unblock the thread. The specific action taken depends on the ODI implementation. The ODI might not always succeed.

When timer processing calls an ODI, the runtime issues message BBOJ0113I to indicate that timer processing has intervened in the processing of a request. The message includes a request identifier, which you can compare with other timeout-related messages and display-command results. Typically, message BBOO0327I is also issued when the controller cleans up the request.

Eventually, the servant stops trying to interrupt the request. Why should this happen? Imagine the following sequence of events:

1. Application performs some operation that causes an ODI to be registered
2. Application hangs
3. Servant timer "pops" (expires) and timer processing finds and drives the registered ODI
4. ODI wakes up the thread, which then causes the ODI to be deregistered.
5. Application receives an exception from the operation (a likely occurrence).

Here, instead of terminating with an error, the application drives the operation again. If the operation hangs again, this process will repeat continuously. Eventually WebSphere detects this condition and "gives up" — that is, it stops trying to unblock the request.

Giving up on a blocked request

Before we discuss the ODIs provided in WebSphere Version 7, let's look at what happens if the servant gives up on a blocked request or WebSphere runs out of ODIs to call.

Gathering documentation

The first thing to do when a servant gives up on a blocked request is to gather some documentation (a dump) about the request. You can have WebSphere request the dump by specifying the variable **server_region_*xxxx*_stalled_thread_dump_action***,* where *xxxx* indicates the type of request: *http*, *iiop*, *mdb*, *https*, *sip*, or *sips*.

Valid settings for this variable are:

SVCDUMP	Requests an SVCDUMP of the servant region
JAVACORE	Collects the call stacks for all Java threads and other information into a file
HEAPDUMP	Collects the JVM heap into a file
TRACEBACK	Captures the call stack of the dispatch thread to the error log (usually the SYSOUT DD card)
JAVATDUMP	Calls a JVM method to create a TDUMP
NONE	No documentation is collected.

After the dump is taken, the controller is notified of the unresponsive request. The controller cleans up the request and responds to the client if appropriate. Then, the controller determines whether to abend the servant to end the timed-out request.

Setting the stalled thread threshold

To determine whether to abend the servant, the controller checks a new configuration variable called **server_region_stalled_thread_threshold_percent**. This variable specifies the percentage of servant dispatch threads (0 to 100) that can stall before the controller abends the servant. A *stalled request* is defined as one that has become unresponsive after its dispatch time has expired and timer processing has stopped trying to unblock the request.

For example, a value of 50 means that half of the dispatch threads in the servant must be stalled before the controller abends the servant. To have the controller abend the servant the first time it abandons a blocked request, set this variable to zero (the default).

The servant uses a counter to track the number of stalled threads. If a request does eventually finish on its own, the count of stalled threads is decremented.

Use this option with care. If you can live with a few hung dispatch threads, consider setting the stalled thread threshold above zero. Before doing so, however, consider whether having some number of stalled threads would pose any consequences to the server. After all, each hung thread is one less thread for dispatching requests.

You might prefer that cost to having to restart the whole servant. If the controller abends the servant, the existing options we described in IBM White Paper WP101233, such as "servant-survivor" and "timeout delay," still apply.

ODIs provided in Version 7

One ODI is registered automatically at the start of dispatch for every request. We call it the JVM ODI because it makes use of the class **com.ibm.jvm.InterruptibleThread** *isBlocked( )* and *unblock( )* methods.

The JVM ODI is the last resort if no other ODI is registered, or if WebSphere is unsuccessful with the other registered ODIs. The *unblock( )* method can break a thread out of certain Java functions with an appropriate exception. For example, a *read( )* on a Java Socket can be interrupted with an *IOException* by the *unblock( )* method. This action allows WebSphere to interrupt an application that has opened a TCP/IP connection to some remote device and has timed out waiting for a response. This scenario should include any Type 4 connector to a resource manager.

WebSphere provides additional ODIs for some common hang scenarios in which it might be possible to unblock the dispatch thread. Additionally, type 2 connectors to local resource managers can implement an ODI to help break free from hangs.

We might add more ODIs over time as we find new scenarios to address.

Controller timer is the failsafe

With the new servant timer in WebSphere Version 7, you might wonder what purpose is served by the existing, controller timer, which is set for "slightly longer than the dispatch timeout?"' This timer handles the situation in which a misbehaving application prevents the servant timer from working.

The controller timer expires well after either the request should have ended or the runtime code in the servant has notified the controller of the stalled request. If this excessive amount of time passes with no response from the servant, the controller begins processing to abend the servant, as if the stalled thread threshold was reached.

Because incrementing and decrementing the count of stalled threads is performed by the servant, the expiration of the controller timer indicates that something is wrong in the servant and we cannot trust it anymore. The controller timer is considered the "failsafe" timer; it should normally not receive control.

What about protocol_http_timeout_output_recovery=session?

Previously, you set the variable **protocol_http_timeout_output_recovery**, as follows:

- When set to *servant,* a dispatch timeout causes the controller to respond to the client and abend the servant region.
- When set to *session,* the controller only responds to the client. The request is allowed to run to completion without being timed. This option allows the dispatch timeout to be used as a response-time value, without having any consequences for the dispatched request. You should use this option only if you are sure that the request will end eventually and you do not care how long it takes to do so.

How does this option interact with the new timeout processing in WebSphere Version 7? The same way. If you set output recovery to *session*, the dispatch timeout triggers a response to the client and nothing else is done. No attempt is made to interrupt or interfere with the dispatched request. Essentially this option disables all of the function we have been describing so far.

If you have previously set output recovery to *session* to avoid servant abends for timeouts, consider changing the value to *servant*, which is the default. The new timeout function might end problem requests without an abend, depending on the cause for requests taking longer than usual to complete.

Using the CPU timer to avoid excessive CPU

One scenario described in IBM White Paper WP101233 is that of a looping request that uses CPU until a timeout abends the servant. For this problem, the usual recommendation is to set the timeout as low as reasonably possible to end the loop quickly and save CPU time. The new timeout functions we described in previous sections do not apply in this scenario (an ODI cannot interrupt a tight loop).

WebSphere Version 7, however, adds a CPU timer, which can help to prevent the dispatched thread from using excessive CPU.

How does the CPU timer work?

If a request exceeds its CPU time limit, the CPU timer stalls the thread and notifies the controller. CPU timer processing issues message BBOJ0114I, which includes the same request identifier used in other timeout related messages.

Depending on whether you specified a stalled thread threshold, this action either causes the servant to be abended immediately or delayed for some period. In any event, because we cannot interrupt the loop, the controller must abend the servant to end the dispatch of the request.

Before you conclude that the CPU timer is the solution for excessive CPU usage by runaway WebSphere applications, let's look at how this function works. Though useful, the CPU timer is not applicable to every situation involving excessive CPU usage.

CPU timer considerations

To monitor CPU usage for the dispatch thread, WebSphere uses the *setitimer* API provided by z/OS UNIX System Services. This API requests that a signal be sent to the dispatch thread if it exceeds the CPU time specified on the variable **server_region_request_cputimeused_limit**.

Here, we encounter the first limitation of using the CPU timer. Depending on what the application is doing when the CPU timer expires, the dispatch thread might not receive the signal. For example, if the application has made a JNI call to native code and has issued a PC instruction, the signal cannot be delivered immediately. Rather, the signal must wait for the thread to enter a receptive state (for example, return from the PC routine).

Therefore, if a loop occurs in a PC routine, the CPU timer does not help. However, if the application is looping around a call to a service that makes the thread unreceptive to signals, the signal is delivered when the thread returns from the call.

After the signal is delivered, the servant does three things:

1. **Gathers diagnostics.** To determine which documentation to gather, the server checks the variable **server_region_cputimeused_dump_action**. The options are those we listed previously for the variable **server_region_http_stalled_thread_dump_action**. By default, the servant collects the call stack for the dispatch thread.

2. **Lowers the priority of the dispatch thread**. Every dispatch thread is associated with a WLM enclave. The servant uses the WLM IWMERES QUIESCE function to lower the enclave priority to the bottom (below discretionary work).
3. **Notifies the controller**. The servant notifies the controller that the request is stalled. This action causes the controller to clean up the request, respond to the client, and increment the stalled thread count.

Some things to note about CPU timer processing:

- Lowering the priority of the thread might not prevent it from running. As long as CPU resources are available (the processor is not running at 100% utilization), the application continues to use CPU. This situation might be common on a test system in which a limited workload is running.
- The stalled thread count is incremented. Normally, this count can be decremented if the request completes. Though possible, it is unlikely that a request that has exceeded its CPU time limit will complete. Therefore, if the CPU timer expires for a request, the servant region will probably be abended.

Advantages of the CPU timer

Given these disclaimers, the CPU timer offers these advantages over the existing dispatch timer:

- The dispatch time monitors elapsed time. You might be more willing to accept excessive elapsed time for a request than excessive CPU time. If so, you can set the CPU timer to a much lower value than the dispatch timer. You might, for example, want a 60-second dispatch timer, but allow no more than one or two seconds of CPU time.
- The CPU timer gives you an opportunity to gather diagnostics as soon as a request uses too much CPU, rather than later when it has been spinning until the dispatch timer ran out.
- In a heavily used system, quiescing the enclave stops the request from running, which is exactly what you want.
- If you know a request that uses too much CPU will be caught by the CPU timer, you can use the stalled thread threshold to control how many runaway WebSphere applications will be caught before the servant is abended. Remember, if you are not certain whether the CPU timer will affect a particular runaway application, be careful with the stalled thread threshold.

Using the new MODIFY,DISPLAY, THREADS command

WebSphere Application Server on z/OS supports a number of useful forms of the command MODIFY DISPLAY. We examined some of the more interesting varieties in IBM White Paper "WebSphere z/OS V6.1 - Hidden Gems and Little Known Features Hidden Gems," WP101138.

In its basic form, the MODIFY DISPLAY command tells you the name of the application server and its code level: `MODIFY <server>,DISPLAY.`

In Version 7, we added a new sub-option called THREADS. This command, which offers both a SUMMARY and DETAILS format, displays information about server activity, which you can use for identifying requests that run longer than expected.

SUMMARY output

The base command is `MODIFY` *`server`*`,DISPLAY,THREADS`. This command displays the status of the servant region for each currently dispatched request (one line of output for each active request). Not shown are requests that are queued, waiting to run.

The following sample shows the 'SUMMARY' format output. To request this format, you can specify the SUMMARY keyword on the command (usually, SUMMARY is the default).

```
BBOJ0111I:  REQUEST   ASID JW TO RE DISPATCH TIME
BBOJ0112I: ffffe453 0X0041  Y  N  N 2008/04/14 18:38:12.391628
BBOJ0112I: ffffe452 0X0041  N  N  N 2008/04/14 18:38:27.473191
BBOJ0112I: ffffe454 0X0041  Y  N  N 2008/04/14 18:38:12.319306
BBOJ0112I: ffffe451 0X003C  N  N  N 2008/04/14 18:38:27.485103
```

In the command output:

- The `REQUEST` column identifies the request. This identifier appears in various timeout related messages.
- The `ASID` column provides the address space identifier (ASID) of the servant region in which the request was dispatched.
- The next three columns of the output indicate the status of the request, as follows:
  - JW shows whether the JVM has indicated that this thread is currently in a wait, as defined by the '*isBlocked( )*' method mentioned previously. A 'Y' in this column does not necessarily mean that the request is stalled; it might just be waiting for something.
  - TO indicates whether the request has Timed Out. A 'Y' in this column indicates that WebSphere has attempted to unblock the request.
  - RE indicates whether WebSphere has stopped trying to unblock the request (the Retry count is Exceeded).
- The final column shows the time when the request began dispatch in the servant region.

Naturally, when this output is displayed, the requests might be long gone. In normal processing, requests might be running at hundreds or thousands per second. By the time the WTOs are issued, the requests they describe are probably completed.

Thus, the filters described in the next section show you all of the long running requests, or show the absence of long-running requests.

New filters for the MODIFY command

Because you are probably more interested in identifying timed-out requests or stalled requests, we added a number of filters to the DISPLAY,THREADS command.

AGE=

This filter displays the requests that have been in dispatch longer than the specified amount of time (in seconds). For example:

```
MODIFY <server>,DISPLAY,THREADS,AGE=30
```

TIMEDOUT

This filter displays timed out requests (those having a ‘Y’ in the TO column). This is an easy way to see all of the blocked requests.

```
MODIFY <server>,DISPLAY,THREADS,TIMEDOUT
```

ALL

This filter shows all requests. This is the default.

ASID=

This filter shows all of the requests for the specified address space identifier (in hex) of a particular servant region. For example:

```
MODIFY <server>,DISPLAY,THREADS,ASID=41
```

Combining keywords

You can also combine keywords, like this:

```
MODIFY <server>,DISPLAY,THREADS,TIMEDOUT,ASID=41
```

DETAILS command syntax

The previous sample is the ‘SUMMARY’ output, which you can specify with the SUMMARY keyword (the default, usually). For more information, you can specify the DETAILS keyword.

You probably do not want the details for all of the requests currently in dispatch (although you can do that). The DETAILS keyword is most usefully used with some of the filters described above.

Some examples:

```
MODIFY <server>,DISPLAY,THREADS,TIMEDOUT,DETAILS
MODIFY <server>,DISPLAY,THREADS,AGE=30,DETAILS
```

With the REQUEST filter, you can get detailed information about a specific request. On REQUEST, you specify the request ID, which is shown in the summary report.

Because you are likely to get the request ID from the SUMMARY display, SUMMARY is not the default for this command. Therefore, you can enter the command like this:

```
MODIFY <server>,DISPLAY,THREADS,REQUEST=FFFFE4F3
```

If you really just want the summary line you can, of course, add the SUMMARY keyword to get that.

DETAILS output

Let's look at some sample output from the preceding command:

```
     BBOJ0106I: REQUEST ffffe453 ASID 0X0041 TCB 0X008CAE88
     BBOJ0119I: CONTROLLER RECEIVED REQUEST AT 2008/04/14 18:38:12.391478
     BBOJ0120I: CONTROLLER QUEUED REQUEST TO WLM AT 2008/04/14 18:38:12.391522
     BBOJ0107I: SERVANT DISPATCHED REQUEST AT 2008/04/14 18:38:12.391628
     BBOJ0108I: JVM THD IS HUNG: ITI INACTIVE
               BBOJ0110I: DETAILS FOR JVM INTERRUPTIBLE THREAD: Monitor ACTIVE
```

The first line (message BBOJ0106I) tells us:

- Request identifier
- ASID of the servant region in which the request is dispatched
- Address of the MVS task control block (TCB) for the dispatched request.

If you get a Java dump or just dump the call stacks of all the threads in the servant, this information will help you find the right thread.

The next three lines (BBOJ10119I, BBOJ0120I, and BBOJ0107I) are the important times for this request:

- When the controller first saw the request
- When the request was put on the queue waiting to dispatch
- When the servant got the request from the queue to begin dispatch.

The fifth line (BBOJ0108I) provides the JVM evaluation of the thread and whether something is being done in response. JVM THD IS HUNG means that JVM thinks the thread is waiting (*isBlocked( )* returned true). This message corresponds to a ‘Y’ in the JW column in the summary display.

The ITI INACTIVE part of the message indicates that WebSphere has not yet tried to interrupt the request (this corresponds to an ‘N’ in the TO column in the summary display). This field can also be ACTIVE, meaning that WebSphere is interrupting the request (corresponding to a ‘Y’ in the TO column in the summary display), or ENDED which means WebSphere has stopped trying to unblock the request (corresponding to a ‘Y’ in the RE column in the summary display).

By the way, ITI stands for InterruptibleThreadInfrastructure — the part of WebSphere that manages this timeout processing.

The last line (BBOJ0110I) is the extra information in the details display. The display command asks each registered ODI what the ODI would like to say. In this case the only registered ODI is the JVM ODI which can interrupt a Java blocked thread. Because we have already told you whether the JVM considers the thread blocked, there is not a lot more for this ODI to add. It just says Active to let you know it is present and was hooked in properly. Other ODIs that cover real blocking activities can provide more information.

Other ODIs might present different information, for example:

```
BBOJ0110I: DETAILS FOR OTS: URID c25926477e42700000000004201010000
BBOJ0110I: DETAILS FOR GIOP Outbound: Target Operation = resolve_complete_info
```

MBean interface

The code that gathers information for the DISPLAY command has also been made available as a JMX Mbean. This means that you could write a small Java program that connects to the server and collects this same information without having to deal with the MVS console and parsing the resulting message.

The name of the MBean is InterruptibleThreadInfrastructure and the operation is getAll. A call returns a ThreadInformation object, which contains information in ThreadDetails objects about all of the dispatch threads.

Using the dispatch progress monitor

The occasional long running request represents a problem. The request might not even timeout, it just takes too long. How does one identify them? In checking the RMF report, you might notice some requests that exceed the goals set for the workload. A long running request can be so infrequent, however, that delay monitoring and other techniques simply do not see them because they are averaged out of the results.

Turning on tracing is not a solution, either. In the flood of trace output from hundreds or thousands of requests, it is difficult to find traces related to the occasional long running request. In fact, if problem requests are sufficiently infrequent, one might not even occur while tracing is turned on.

To help address this problem, WebSphere Version 7 introduces the dispatch progress monitor (DPM).

The DPM works like this:

1. At initialization the DPM is disabled; you can use the MODIFY command to enable the DPM for a particular protocol by specifying an interval.
2. When a request for that protocol begins dispatch in the servant, a timer is set for the specified interval.
3. When the timer expires and the request has not finished dispatch, timer processing issues a message to the error log (BBOJ0118) with information about the request and gathers documentation based on the configuration.
4. The timer is set for the configured interval value again.

To enable the DPM, enter the MODIFY command, as follows:

MODIFY <*server*>,DPM,*xxxx*=<*interval*>

Where:

- *server* is the name of the server
- *interval* is the interval value you want the DPM to use (zero to 255 seconds).
- xxxx is the type of request: HTTP, IIOP, MDB, HTTPS, SIP, or SIPS.

To disable all DPM monitoring, enter this command:

MODIFY <*server*>,DPM,CLEAR_ALL

You can specify which documentation is gathered when the DPM timer expires through the environment variable **server_region_dpm_dump_action**.

Valid settings for this variable are:

SVCDUMP	Requests an SVCDUMP of the servant region
JAVACORE	Collects the call stacks for all Java threads and other information into a file
HEAPDUMP	Collects the JVM heap into a file
TRACEBACK	Captures the call stack of the dispatch thread to the error log (usually the SYSOUT DD card)
JAVATDUMP	Calls a JVM method to create a TDUMP
NONE	No documentation is collected.

The default is TRACEBACK.

You can dynamically change the dump action with the DPM MODIFY command, for example:

MODIFY <*server*>,DPM,DUMP_ACTION=JAVACORE

Your setting for this variable takes affect immediately. Of course, if a request is dispatched with the DPM turned off, no timer will expire to notice the change.

Example of using DPM

Suppose that your application consists of servlets driven by HTTP requests, most of which complete dispatch in less than a second. Occasionally, a request takes longer, say 10 or 20 seconds. To catch these long running requests, you can set the DPM HTTP interval to five seconds. This setting yields a traceback snapshot a few times during the dispatch of these longer running requests. Now you have something to help you figure out why these requests take longer.

Be careful, however, not to set this interval too low, thus catching normal requests. If most of your other requests complete in five or six seconds, setting a DPM interval of five seconds might cause a large number of stack traces (one for nearly every request).

If you are not sure how the DPM is configured, you can enter this display command to find out:

MODIFY server,DISPLAY,DPM

This command displays output like this:

```
BBOO0361I DISPATCH PROGRESS MONITOR (DPM) SETTINGS:
IIOP(000):HTTP(010):HTTPS(015):MDB(000):SIP(020):SIPS(000),DUMP_ACTION
(JAVATDUMP)
```

What other information about the request do you get in message BBOJ0118? Here is an example with the dump action set to TRACEBACK (and formatted for readability):

```
BBOJ0118I: ThreadDetails: ASID = 005B, TCB = 0X008CBE88, Request = fffff503,
Is JVM Blocked = false, Tried to interrupt = false, Given up = false,
Internal Work Thread = false, Hung Reason = Not Hung,
SR Dispatch Time = 2008/05/05 12:15:31.371625,
CTL Receive Time = 2008/05/05 12:15:31.366693,
CTL Queued to WLM Time = 2008/05/05 12:15:31.371328,
Details = , ODI Details = .JVM INTERRUPTIBLE THREAD, Monitor ACTIVE.

BBOJ0117I: JAVA THREAD STACK TRACEBACK FOR THREAD WebSphere:ORB.thread.pool
t=008cbe88: Dispatch Progress Monitor
Traceback for thread WebSphere:ORB.thread.pool t=008cbe88:
com.ibm.ws390.orb.ClientDelegate.invokeRequestCFW(Native Method)
com.ibm.ws390.orb.ClientDelegate.commonInvoke(ClientDelegate.java:998)
com.ibm.ws390.orb.ClientDelegate.invoke(ClientDelegate.java:845)
org.omg.CORBA.portable.ObjectImpl._invoke(ObjectImpl.java:484)
com.ejb.test.hello.HelloSecondHome_Stub.create(HelloSecondHome_Stub.java:207)
com.ejb.test.hello.HelloFirstBean.sayHelloOne(HelloFirstBean.java:76)
com.ejb.test.hello.EJSRemoteStatelessSayHelloFirst_67c1d243.sayHelloOne(EJSRem
oteStatelessSayHelloFirst_67c1d243.java:41)
```

Revisiting timeouts from earlier scenarios

The last section of IBM White Paper "WebSphere Application Server for z/OS V6.1 Configuration Options for Handling Application Dispatch Timeouts" (WP101233) described three possible timeout scenarios and provided recommendations for configuring WebSphere to handle them.

Let us review those earlier scenarios to see how the new capabilities in WebSphere Application Server for z/OS Version 7 might affect the outcome.

Defending against timeouts

When you migrate to Version 7, instead of immediately abending the servant region when the dispatch time expires, WebSphere will start nudging the request to try to get it to complete. In the end, WebSphere might still abend the servant region. However, this nudging might allow a request that has been in dispatch for too long to complete with an error, but without making things worse.

One recommendation for this scenario was to verify that your timeout values were set appropriately; this recommendation still applies. However, if you have set the timeouts to larger values to prevent the servant from being abended, you might consider lowering the value if you think WebSphere might "nudge" one to completion. This gives you the benefit of the timeout processing without the potential impact of a servant restart.

How do you know what dispatch times are reasonable for you? You can check RMF reports for the completion times for requests, organized by report class. You might also examine the data provided in the SMF Type 120 Subtype 9 record, which is new in WebSphere Version 7. Timestamps provided in the new record can tell you how much time requests take to complete.

Version 7 also adds the consideration of what documentation you would like to have gathered if the servant region gives up trying to nudge a timed out request. If the servant is eventually abended, you might receive an SVCDUMP with the abend. However, you could also get a simple traceback or javacore, which might be easier to manage and something an application developer is more used to seeing.

Should you set a CPU timeout? Consider it, but remember that the CPU timer causes the servant to be abended when a request uses too much CPU. Be careful not to set too low of a value. Again, SMF records can tell you how much CPU is being used.

Application hang timeouts

Application hang timeouts are usually either hangs waiting for external resources (another server, for example) or deadlocks or contention in the server. In most of these cases, the new processing can unblock a request so that it completes without more severe consequences (an abend).

Here, the recommendations provided in IBM White Paper WP101233 are still valid. Depending on the nature of the hang, consider setting the stalled thread threshold to a non-zero value to allow a few hangs to accumulate before abending the servant.

Use this option with care. If you have other problems, especially a runaway request that is looping, setting the stalled thread threshold could prevent WebSphere from ever abending the servant. Be sure you understand why the requests are timing out before you allow multiple hangs to accumulate.

Runaway application timeouts

There was not much to be done in this scenario in earlier releases. IBM White Paper WP101233 mainly advised you to avoid doing anything to delay the abend of the servant. However, with Version 7 you can now specify a CPU timeout. As noted earlier there are some cases where this timeout might not actually do anything. However, if your application is looping in Java code, you can set the CPU timeout which will, at the very least, cause the servant to be abended. This lets you set a dispatch timeout of perhaps several minutes, while only allowing seconds of CPU to be used. With just a dispatch timer you could potentially consume a lot of CPU before running out of elapsed time.

If you think the CPU timer will catch your looping request and you are running on a heavily loaded system, you might even consider setting the stalled thread threshold to cause CPU-timed-out requests to build up before the servant gets abended. But remember, just because we have made the request a very low priority does not mean the thread will not still get dispatched. It can still consume CPU if any is available.

Conclusion

WebSphere Application Server for z/OS Version 7 adds useful new options and services for avoiding timeout related abends.

We hope that this IBM White Paper helps you make the best use of these new options in your environment.

Related reading

You can find this IBM White Paper and the following documents on-line at the Techdocs Web site: `www.ibm.com/support/techdocs`

Document Title	**Document Number**
WebSphere Application Server for z/OS V6.1 Hidden Gems and Little Known Features	WP101138
WebSphere Application Server for z/OS V6.1 Configuration Options for Handling Application Dispatch Timeouts	WP101233
Understanding SMF Record Type 120, Subtype 9	WP101342

Document change history

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

December 6, 2008	Original Version
December 9,2008	Document number WP101374 applied to the document and republished.
May 2, 2013	Fix errors in MODIFY DPM section per comments from Magda M.

End of WP101374