WebSphere Application Server for z/OS V8

Granular Timeout Controls, Request-Level Tracing, Message Tagging and More!

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102023** under the category of "White Papers"

Version Date: December 3, 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

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

Edited by Don Bagwell
IBM Advanced Technical Skills
Gaithersburg, MD

Many thanks go to ...The zRuntime team (Dan Gritter, Dave Sudlik, Mike Ginnick, Gary Picher, Junior Ricon, Ken Daniels, Pat Cupka, Rob Alderman, Tim Spewak, Surya Pericherla, and Tom Henry) for getting all this work done and to the WSC, especially Don Bagwell, Mike Loos, John Hutchinson, and, of course, Mike Cox.

Table of Contents

Introduction

Customers who have WebSphere Application Server running on distributed platforms often only deploy a single application per server. If they move these applications to z/OS, or if they deploy new applications on z/OS, they often tend to follow this same pattern. As the number of applications grows on distributed this can lead to a lot of servers and, potentially, a lot of boxes to run them. One of the advantages of z/OS is the ability to run numerous application servers on a single OS image. However, running hundreds of application servers on a single z/OS LPAR uses a lot of resources and can be quite expensive.

There might be business or other reasons why you have to separate applications into different servers. But often the reason is simply because “That's the way we do things.” With whole servers dedicated to applications which are lightly used, this can mean a lot of unnecessary resource usage and complexity of configuration and management.

Therefore, as we looked to Version 8 we tried to consider what issues customers might face consolidating multiple applications into a single server. The problem that immediately jumped out at us was server-level configuration items that would reasonably vary with different applications. So we sought to increase the granularity of control.

One example of a server-level configuration item was timeouts. In prior versions you could configure a timeout only at the server level. Each protocol got its own dispatch timeout, but that was as granular as it got. Therefore every HTTP request, no matter what it was, was subject to the same timeout. This naturally leads customers to configure very high timeout values which reduces the value of having timeouts at all.

Continuing with this train of thought, we realized that even within a single application you might easily have different timeout requirements. One servlet might normally complete very quickly, perhaps sub-second. Another servlet in the same application might do much more processing and never finish in less than 30 seconds. Even an application-level timeout would have to be set unnecessarily high for the small requests to prevent the larger ones from incorrectly timing out.

What we really need is granularity at the request level. And it's not *just* timeouts. So from that requirement came the function you'll read about in this document.

Overview of the Essential Underlying Function

Fortunately we already had a way to specify a single characteristic of a request at a granularity that reaches down to the individual servlet, EJB method, etc. We used that as the starting point and extended it to provide you greater granularity.

Way back in Version 5.1 we introduced the WLM Classification XML file. This file allowed you to specify a Transaction Class name to be given to the z/OS Workload Manager (WLM) for each request. WLM used this Transaction Class name to classify the request. The server finds the file by looking at the setting of the variable **`wlm_classification_file`**.

This function allowed WAS to associate a specific configuration value to a very granular level: that is, down to specific request identifiers. That meant within a single application different requests could be identified (for HTTP it was done by matching against a URI pattern), then having WAS z/OS assign a configuration value to that very granular level of control.

It was an ideal starting point for the new function we were looking to implement. The existing WLM classification framework could be extended to permit additional options to provide you with granular control to the same low level ... the specific request level.

Review of the WLM classification file

Here is an example[1] of a fragment of a very simple WLM Classification XML file:

```
<InboundClassification type="http" 1
          schema_version="1.0" default_transaction_class="M"> 2
 3 <http_classification_info transaction_class="N"
          host="host.company.com">
    4 <http_classification_info transaction_class="Q"
           uri="/gcs/admin"/>
    5 <http_classification_info transaction_class="R"
           uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

Notes:
- This is just the section of the file for HTTP requests. Other request types such as IIOP or MDB are also supported. To keep things simple we're focusing on HTTP.
- Transaction names such as M, N, Q and R are not good examples. We use those for the sake of simplicity. More meaningful names should be used in general practice.

Let's look at how the defaults work and how the right transaction class is found. This will be important in examples later on where things get more complicated.

First, note the structure of the sample XML offered above:

1	The request type. In this case HTTP. Options are: `iiop`, `http`, `mdb`, `sip`, `ola` and `internal`
2	The default WLM transaction class in the event none of the more specific patterns match.
3	The first of three specific classification patterns. This specifies the host value of the request.
4	The second of three classification patterns. This specifies a URI pattern.
5	The third and final classification pattern. This specifies a URI pattern with a wildcard asterisk.

The nesting is important. Here are some example inbound requests and how they would match given the example XML from above:

URL	
`http://myhost.com/servlet`	Would receive the default transaction classification of **M** because neither the host nor the URI matches the other patterns.
`http://host.company.com/servlet`	Would receive the transaction classification **N** because it would match with 3 from above but not 4 or 5.
`http://host.company.com/gcs/admin`	Would receive transaction classification **Q** due to matching on *both* 3 and 4. This is an example of why nesting is important to understand.
`http://myhost.com/gcs/admin`	Would receive the default transaction classification **M**. The URI `/gcs/admin` matches 4, but the host does not match 3. This is why we say nesting is important.
`http://host.company.com/gcs/admin/1`	Matches 3 and 5 and thus TC **R**.
`http://host.company.com/gcs/admin/123`	Matches 3 and 5 and thus TC **R**.

1 In the WAS z/OS V8 InfoCenter, search on string `rrun_wlm_tclass_dtd` for the article on the classification file.

Now that we understand the Classification XML file, we can begin to see how this file might be extended to allow you to specify *other* attributes for a request. In addition to specifying the Transaction Class name you could also specify, for example, different dispatch timeout values or other attributes.

Other challenges considered and addressed in WAS z/OS Version 8

In planning the Version 8 release and this new granular control feature, we investigated and addressed several other challenges we knew to be present:

- In WAS z/OS Version 7 and earlier any changes to the Classification XML file required a server restart. In WAS z/OS V8 we provided a MVS `MODIFY` command that will dynamically refresh the read-in XML file. This avoids the need to restart the server to pick up changes or to back out changes you don't wish to use.
- Some of the attributes we wished to put into this new function also had environment variables or other properties that already existed. We knew the new function had to take into account cases where properties set elsewhere were also set in the Classification XML file.
- Some environment variables have MVS console `MODIFY` commands that allow you to dynamically change the value in use by the server. We knew the new function had to take into account cases where properties set in the XML are dynamically altered through MVS `MODIFY`.

The remainder of this document will look at the new tags we have added to the Classification XML file and how they interact with the existing configuration variables and any related Modify commands. We will wrap up with some detail on how you dynamically update the XML being used by the server.

Dispatch Timeouts

Every request that is dispatched in a servant region is covered by a dispatch timeout. The value used for the time limit is normally taken from an environment variable. Which environment variable depends on how the request came into the server. The following table shows which environment variable applies for which types of inbound requests.

Work Type	*Dispatch Timeout Variable Used*
IIOP	`control_region_wlm_dispatch_timeout`
HTTP	`protocol_http_timeout_output`
HTTPS	`protocol_https_timeout_output`
SIP	`protocol_sip_timeout_output`
SIPS	`protocol_sips_timeout_output`
Message Listener Port MDBs	`control_region_mdb_request_timeout`
SIBus or Activation Specification (MDBs from the CRA)	`control_region_wlm_dispatch_timeout`
WOLA	`control_region_wlm_dispatch_timeout`
Other (Message Listener Port in the SR, MBeans, other internal stuff)	`control_region_wlm_dispatch_timeout`

In WAS z/OS Version 8 these timeout values may also be specified in the Classification XML file in addition to environment variables. The precedence relationship is this:

- If *no* value is found in the Classification XML file, then the environment variable value will be use.
- If a value is found in the Classification XML file and it applies to a request, the value in the XML file takes precedence over the environment variable value. An exception is when MODIFY commands are involved. We'll look at those a bit later.

What does the dispatch timeout look like in the XML file? You just add the **`dispatch_timout`** tag to the parts of the XML where you want to set a value.

Go back to our example earlier. Suppose we want to set a dispatch timeout of five minutes (300 seconds) for any HTTP request *except* for that clause for the specific URI `/gcs/admin`. Suppose that's a long running thing and we want to set the dispatch timeout for that one to 30 minutes (1800 seconds). Then our XML fragment would look like this:

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" dispatch_timeout="300">
      <http_classification_info transaction_class="Q"
             uri="/gcs/admin" dispatch_timeout="1800"/>
      <http_classification_info transaction_class="R"
             uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

The nice thing about setting the dispatch timeout in the XML file is that you can set a default for each protocol by specifying it on the outermost `http_classification_info` node as we illustrated. That will take precedence unless a dispatch timeout is specified on one of the inner nodes. We illustrated that case as well.

The same `dispatch_timeout` variable is used for *every protocol* supported in the Classification XML file. This provides a way of avoiding the confusing assortment of timeout environment variables and focus instead on a single variable within the XML file.

Queue Timeouts

Work received by the controller is first placed on a WLM work queue before being dispatched to a thread in a servant region[2]. There are timers that track the time work sits on a queue before being dispatched. These *queue timeout* values are controllable by you.

The queue timeout is specified as a percentage of the total dispatch timeout value. For example, suppose you configure the dispatch timeout to 300 seconds and set the queue timeout percentage to 50%. If the request spends more then 150 seconds (50% of 300 seconds) on the queue waiting to dispatch then the request will time out, be cleaned up and a response sent to the client (if appropriate). Configuring a queue timeout helps avoid work getting into the servant region just as the overall dispatch timer is about to pop.

How do you configure a queue timeout with environment variables? Depends on what type of work it is, just like dispatch timeouts. It's time for another table:

Work Type	*QueueTimeout Variable Used*
IIOP	`control_region_iiop_queue_timeout_percent`
HTTP	`control_region_http_queue_timeout_percent`
HTTPS	`control_region_https_queue_timeout_percent`
SIP	`control_region_sip_queue_timeout_percent`
SIPS	`control_region_sips_queue_timeout_percent`
Message Listener Port MDBs	`control_region_mdb_queue_timeout_percent`
SIBus or Activation Specification (MDBs from the CRA)	`control_region_wlm_queue_timeout_percent`
WOLA	`control_region_wlm_queue_timeout_percent`
Other (Message Listener Port in the SR, Mbeans, other internal stuff)	`control_region_wlm_queue_timeout_percent`

Once again this is a bit of a mess, and once again it all gets cleaner and simpler if you simply put a default value in the XML file for each protocol.

The tag for queue timeout percent is **`queue_timeout_percent`**. We will update our example to specify a 50% queue percentage for the 1800 second dispatch timeout. Here we go:

```
<InboundClassification type="http"
           schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
           host="host.company.com" dispatch_time="300">
      <http_classification_info transaction_class="Q"
              uri="/gcs/admin" dispatch_timeout="1800"
              queue_timeout_percent="50"/>
      <http_classification_info transaction_class="R"
              uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

Since we did *not* specify a default `queue_timeout_percent` value on the outermost `http_classification_info` node, it means every HTTP request *except* `/gcs/admin` will get the queue timeout percentage from the `protocol_http_queue_timeout_percentage` environment variable, whatever that happens to be set to.

Setting a default in the Classification XML for each protocol keeps everything in one place and makes it easier to determine what value you are getting.

2 See WP101740 at `ibm.com/support/techdocs` for more details on the interaction between WAS z/OS and WLM.

Timeout Dump Action

There is one more set of variables that can now be handled more cleanly and with more granularity in the classification XML file. These are the variables that control what action is taken when the dispatch timer has expired and the code that handles that timeout has given up trying to get the request to complete. It is time for WAS z/OS to gather some documentation for debug.

As before, there are many environment variables that control the dump action for each of the request protocols. Possible values for all the variables are: `NONE`, `JAVATDUMP`, `TRACEBACK`, `HEAPDUMP`, `JAVACORE`, and `SVCDUMP`. The following table summarizes the environment variables:

Work Type	*QueueTimeout Variable Used*
IIOP	`server_region_iiop_stalled_thread_dump_action`
HTTP	`server_region_http_stalled_thread_dump_action`
HTTPS	`server_region_https_stalled_thread_dump_action`
SIP	`server_region_sip_stalled_thread_dump_action`
SIPS	`server_region_sips_stalled_thread_dump_action`
Message Listener Port MDBs	`server_region_mdb_stalled_thread_dump_action`
SIBus or Activation Specification (MDBs from the CRA)	`server_region_iiop_stalled_thread_dump_action`
WOLA	`server_region_iiop_stalled_thread_dump_action`
Other (Message Listener Port in the SR, Mbeans, other internal stuff)	`server_region_iiop_stalled_thread_dump_action`

The tag for the dump action in the XML is **`stalled_thread_dump_action`.**

Here is our example yet again. This time we have extended it to specify `TRACEBACK` as the default for all HTTP requests, and `SVCDUMP` for our `/gcs/admin` special case:

```
<InboundClassification type="http"
           schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
           host="host.company.com" dispatch_time="300"
           stalled_thread_dump_action=”TRACEBACK”>
      <http_classification_info transaction_class="Q"
              uri="/gcs/admin"
              dispatch_timeout="1800" queue_timeout_percent="50"
              stalled_thread_dump_action=”SVCDUMP”/>
      <http_classification_info transaction_class=“R”
              uri=“/gcs/admin/1*”/>
   </http_classification_info>
</InboundClassification>
```

CPU Timeouts

In the Version 7 release of WAS z/OS the ability to set a limit for CPU time used was introduced. A new variable was introduced -- `server_region_request_cputimeused_limit` -- which specified the maximum CPU permitted *for any given request in the servant region*. For V7 the variable was granular only down to the server level. If a thread consumed more than the maximum CPU allowed, the WLM enclave for the thread was quiesced, which put it in a status lower than discretionary. In effect, the thread was starved of CPU and it would sit there doing nothing.

It was a good step forward. However, with multiple applications installed in a server a "runaway" thread in one application might well be "working as designed" for another application.

To remedy this, in WAS z/OS Version 8 the maximum CPU time allowed for a thread, as well as the dump action taken if a thread exceeds the value, was enabled as granular controls in the Classification XML file.

So with WAS z/OS V8 we have the environment variables from before and the new XML tags for more granular control:

Environment Variables	`server_region_request_cputimeused_limit` `server_region_cputimeused_dump_action`
Classification File Tags	`cputimeused_limit` `cputimeused_dump_action`

For different applications, or different pieces of one application, you can easily configure different CPU time limits and different documentation to gather.

Here is our same example, extended yet again:

```
<InboundClassification type="http"
           schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
           host="host.company.com" dispatch_time="300"
           stalled_thread_dump_action=”TRACEBACK”
           cputimeused_limit="30” cputimeused_dump_action="TRACEBACK">
      <http_classification_info transaction_class="Q"
              uri="/gcs/admin"
              dispatch_timeout="1800" queue_timeout_percent="50"
              stalled_thread_dump_action=”SVCDUMP” cputimeused_limit="300"/>
      <http_classification_info transaction_class=“R”
              uri=“/gcs/admin/1*”/>
   </http_classification_info>
</InboundClassification>
```

In this example we specified a generic CPU limit of 30 milliseconds for all applications with a dump action of `TRACEBACK`. But the specific URI `/gcs/admin`, for which the dispatch timeout value is 1800 seconds, the `cputimeused_limit` is set to 300 milliseconds. We will, however, let it inherit the dump action of `TRACEBACK` from the protocol default.

IIOP Outbound Request Timeouts

Suppose your application, whether it is a servlet, an MDB, or something else, drives an *outbound* request to an EJB located in another server. That request will turn into a CORBA IIOP flow to the other server and your application will wait for a response. The dispatch timeout we configured earlier will keep it from waiting forever. But it would be nice to be able to configure a timeout for this outbound request. That way your application could get control back sooner and take action rather than wait for the whole dispatch timeout interval to expire.

For example, suppose a servlet gets control and has a dispatch timeout of 30 seconds. Suppose that application makes four separate calls to an EJB located on some other server. Normally those remote EJB calls should complete in less than two seconds. Rather than wait for 30 seconds to elapse to give up on the whole request, it would be nice to have a timeout on each outbound request set to, for example, five seconds.

You can configure just this sort of timeout value in Version 7 and before using the property **`com.ibm.CORBA.RequestTimeout`**. This property applies to all outbound CORBA IIOP requests from this server to any other server.

In Version 8 we now allow you to configure an outbound IIOP request timeout in the classification XML. That means that for any specific dispatched request you can control how long it will wait for a response on any outbound requests it makes.

For example, imagine you have two servlets that each drive an EJB request to the same back-end server:

- Servlet A drives a very simple EJB in the back-end server which you know normally takes only a second or two two respond
- Servlet B drives a more complex EJB in the same back-end server, and from experience you know that EJB may take much more time to complete its work and respond

In this example we see Servlet A and Servlet B will naturally have different overall completion times. We saw earlier[3] that the Classification XML file `dispatch_timeout` tag can be used to set this *overall timeout value* to the individual request level.

Another Classification XML tag, new in V8, can be used to control the timeouts for the outbound calls from the servlets to the EJBs. That tag is `request_timeout`.

Why "request" for the outbound call? We chose `request_timeout` for outbound calls because it lined up with the property we are letting you override (**`com.ibm.CORBA.RequestTimeout`**). We realize that variable is somewhat confusing too, but it seemed a good idea to at least be consistent.

We will update the XML we have been building to specify a default outbound request timeout for dispatched HTTP requests and also a specific outbound request timeout for the servlet at `/gcs/admin`. We will use 5 seconds for the default timeout and 30 seconds for the timeout for our special servlet.

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" dispatch_time="300"
          stalled_thread_dump_action="TRACEBACK"
          cputimeused_limit="30 cputimeused_dump_action="TRACEBACK"
          request_timeout="5">
      <http_classification_info transaction_class="Q"
             uri="/gcs/admin"
             dispatch_timeout="1800" queue_timeout_percent="50"
             stalled_thread_dump_action="SVCDUMP" cputimeused_limit="300"
             request_timeout="30"/>
      <http_classification_info transaction_class="R"
             uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

Timeout Recovery Action

Sometimes you do not want a `dispatch_timeout` to stop a runaway or a hung request. You know the request will finish eventually, so what you *really* want is a timer that controls when to reply back to the client. It is okay if a dispatch thread is held up for a while longer, but the client needs to get a response.

In Version 7 there was a set of server variables that controlled the recovery action WAS z/OS was to take when a timeout occurred:

```
protocol_http_timeout_output_recovery          SERVANT | SESSION
protocol_https_timeout_output_recovery         SERVANT | SESSION
protocol_sip_timeout_output_recovery           SERVANT | SESSION
protocol_sips_timeout_output_recovery          SERVANT | SESSION
```

3 See "Dispatch Timeouts" on page 7.

From the InfoCenter: "Specifying `SERVANT` allows for the termination of servants with an `ABEND EC3 RSN=04130007` when timeouts occur, and the HTTP request and socket are then cleaned up. A setting of `SESSION` only cleans up the HTTP request and socket, but no attempt to abend the servant is made."

Note: This option is only available for HTTP/HTTPS and SIP/SIPS requests because IIOP request flow does not allow a client to get a response while the request is still running. It confuses the transactional logic. MDBs do not really even have a response so the option just does not apply in any of the MDB flows (your application might put a message on a response queue, but the server itself has no response to send).

Since these variables are defined at the server level, they apply to all the servlets/JSPs within the applications installed in the server. With WAS Version 8 you can now make this choice at any level from the server through application-level and down to individual servlets/JSPs. To do this you set the XML tag `timeout_recovery` to either “servant” or “session”.

We extend our example XML yet again to show these new options. We will set a default value of “servant' for the outermost `http_classification_info` tag to align with the default of the server environment variables. This provides a way to eliminate ambiguity -- the variables are set in the XML so we know for certain what the values are, rather than wondering if they're set elsewhere. To illustrate the the use of this new tag on a more granular level we will set a value of “session” for the `/gcs/admin/1*` set of URIs.

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" dispatch_time="300"
          stalled_thread_dump_action=”TRACEBACK”
          cputimeused_limit="30 cputimeused_dump_action="TRACEBACK"
          request_timeout="5" timeout_recovery="servant">
      <http_classification_info transaction_class="Q"
             uri="/gcs/admin"
             dispatch_timeout="1800" queue_timeout_percent="50"
             stalled_thread_dump_action=”SVCDUMP” cputimeused_limit="300"
             request_timeout="30"/>
      <http_classification_info transaction_class=“R”
             uri=“/gcs/admin/1*” timeout_recovery="session"/>
   </http_classification_info>
</InboundClassification>
```

Dispatch Progress Monitor (DPM)

The Dispatch Progress Monitor (or DPM) is a function introduced in Version 7. It allows you to configure an interval and an action. Any request which takes longer than the interval to complete will be subject to the action (some documentation is gathered) at that interval.

Note: The DPM function introduced in Version 7 was very handy, but not as granular as we would like. So in Version 8 we provided Classification XML file tags to set the DPM interval and action to a lower level of granularity. What follows is first an explaination of the existing DPM settings, then we'll explain the new V8 Classification XML settings.

For example, if you configure the DPM with an interval of 30 seconds and an action of `TRACEBACK` any request that takes more than 30 seconds to dispatch will get a callstack dump at 30 seconds and at every thirty seconds after that if it keeps running.

There are two parts to configuring the DPM:

- The first part is the *action* to take. You can define that through the following environment variable:

  `server_region_dpm_dump_action`

The value can also be changed dynamically using the following `MODIFY` command:

```
F <server>,DPM,DUMP_ACTION=<action>
```

Your choices of dump actions are the same as for the timeout related actions -- `NONE`, `JAVATDUMP`, `TRACEBACK`, `HEAPDUMP`, `JAVACORE`, and `SVCDUMP`. This action applies any time a request hits the DPM interval. (You can also specify `DUMP_ACTION=RESET`, but we will get to that later).

- The second part is the *interval* to be used. The DPM interval can not be configured through an environment variable. You must set it dynamically using another DPM `MODIFY` command. To set it you specify the protocol whose interval you want to set (for example, HTTP, IIOP, etc.) and the interval you want in seconds. For example:

  ```
  F <server>,DPM,HTTP=30
  ```

  This sets the DPM interval for all HTTP requests to 30 seconds. Only one interval is allowed per protocol for that server.

This is clearly a case where more granularity would be useful. In Version 8 we introduced new tags to define the DPM interval and action in the WLM Classification file. In the XML file you can now specify the tags `dpm_interval` and `dpm_dump_action` for any entry in the file. This enables you to set different DPM intervals for different requests that you may be having trouble with and also collect different documentation based on the nature of the problem.

Let us modify our XML file slightly to set a DPM interval for our specific URI. Since the dispatch timeout we set earlier is 1800 seconds, we probably want a DPM interval that is pretty large as well. Suppose most requests for `/gcs/admin` complete in 500 seconds. We could tell the DPM to capture a traceback for the dispatched request if it takes 1000 seconds. Since the dispatch timeout is 1800 seconds we will only get one hit of the DPM before the dispatch timer pops (since the second DPM interval hit would be at 2000 seconds). Here is the XML:

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" dispatch_time="300"
          stalled_thread_dump_action="TRACEBACK"
          cputimeused_limit="30 cputimeused_dump_action="TRACEBACK"
          request_timeout="5" timeout_recovery="servant">
      <http_classification_info transaction_class="Q"
             uri="/gcs/admin"
             dispatch_timeout="1800" queue_timeout_percent="50"
             stalled_thread_dump_action="SVCDUMP" cputimeused_limit="300"
             request_timeout="30"
             dpm_interval=″1000″ dpm_dump_action="TRACEBACK"/>
      <http_classification_info transaction_class="R"
             uri="/gcs/admin/1*″ timeout_recovery="session"/>
   </http_classification_info>
</InboundClassification>
```

As we will see later, it is possible to dynamically activate a new or updated Classification XML file. It's a very powerful feature but if you're not careful you can introduce unintended problems. Let's explore a hypothetical scenario and describe what's available to recover from configuration problems related to DPM.

Suppose we wanted to add `dpm_interval` to our XML file and activate it dynamically. But suppose we made a terrible mistake and instead of setting the DPM interval to 1000 seconds we set to to just 100. Since we stated earlier we believe almost every request of this type takes 500 seconds, the DPM will hit 5 times for *every* request and collect a callstack. That is probably not what we want. Once we realize there is a problem, how do we stop it?

The obvious answer is to go edit the XML file and make the correction and activate the updated file. That might take a little while. Perhaps longer than you have. What we need is a quick way to just turn the DPM off until we can figure out what went wrong.

In Version 7 you could specify `CLEAR_ALL` as an option on the `MODIFY` command.

```
F <server>DPM,CLEAR_ALL
```

This command will reset all the DPM intervals to zero and sets the dump action to `NONE`. Setting the intervals to zero effectively stops new requests from setting DPM timers. Setting the action to `NONE` turns off any action being taken for DPM timers that may be firing for any active requests.

What does that `MODIFY` command do for Version 8 where the intervals and action are defined in the XML file as well? It does exactly what you would expect: it turns off all the DPM processing immediately.

So if you are experimenting with DPM trying to debug a problem and things start to go bad (for example a dump frenzy), then be ready to issue the `CLEAR_ALL` and shut it down.

Let's explore another hypothetical scenario. Suppose you updated the XML file and activated it dynamically (again, we'll get to how you do that a little later). Suppose you added in some DPM intervals and actions. And suppose you made a mistake and suddenly there's a frenzy of javacores or some other action. Ooops ☺. So you quickly issue the `DPM,CLEAR_ALL` and stop it. You go back to the XML file and find the mistake and correct it. You dynamically activate the new XML file. But the DPM is ignoring the XML file settings[4].

How do you reset the DPM to go back to its normal behavior? By using the `RESET` option:

```
F <server>,DPM,RESET
```

That command sets the DPM back to its 'normal' behavior of honoring the XML file settings or, if you don't have an XML file, back to the environment variables it started with.

Use the `DUMP_ACTION=RESET` command to go back to the XML file settings for just the dump action,like this:

```
F <server>,DPM,DUMP_ACTION=RESET
```

There is one more modify command that is like `CLEAR_ALL` but not quite. It is the `INTERVAL` option on the DPM `MODIFY` command. You use `INTERVAL` to set the DPM interval for all the protocols at once. So instead of issuing a `MODIFY DPM,IIOP=0` and a `MODIFY DPM,HTTP=0` and so forth, you can just `MODIFY DPM,INTERVAL=0` and set them all at once. This action will also override the XML file values. It is different from `CLEAR_ALL` because it does *not* change the DPM action to `NONE`. The action stays set to whatever the current value is.

How do you turn that off and go back to the XML values? `RESET` is the key again. You can specify `INTERVAL=RESET` and the DPM will go back to honoring the DPM intervals from the XML file. You can also specify `RESET` as a value for any of the individual protocols. For example, you *can* issue a `MODIFY DPM,HTTP=0` to set the HTTP DPM interval to zero and then later issue a `MODIFY DPM,HTTP=RESET` to go back to using values from the HTTP section of the XML file.

It is a little complicated. The simplest approach is to use the XML file to carefully set the DPM interval and action for the specific request you are interested in. When you activate the XML, be ready to issue the `CLEAR_ALL` command in case there is a problem. Then use `RESET` to go back and try again when you've fixed the XML.

The `MODIFY DISPLAY,DPM` command helps you determine the current state of DPM. The format of the command is:

4 Because of a previously issued MODIFY command.

```
F <server>,DISPLAY,DPM
```

And the output looks like this:

```
BBOO0361I DISPATCH PROGRESS MONITOR (DPM) SETTINGS:
IIOP(RESET):HTTP(RESET):HTTPS(RESET):MDB(RESET):SIP(RESET):SIPS(RESET)
:OLA(RESET) DUMP_ACTION(RESET)
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,DPM
```

SMF Recording

Starting with WAS z/OS Version 7, WebSphere can write SMF Type 120 Subtype 9 record for every request it processes. You can control whether or not these records are written by setting the environment variable `server_SMF_request_activity_enabled`. The 120-9 record has some sections which are optional. You can enable or disable those sections based on the settings of three related environment variables:

- `server_SMF_request_activity_CPU_detail`
- `server_SMF_request_activity_timestamps`
- `server_SMF_request_activity_security`

Additionally, you can use the `MODIFY` command `SMF,REQUEST,ON` or `OFF` to turn the recording on or off. Additional options on the SMF `MODIFY` command allow you to dynamically turn on and off the optional sections.

In Version 8 we have added tags to the Classification XML file to allow you to control what SMF data is recorded at a protocol or even at an individual request level. The new tag is `SMF_request_activity_enabled` and the value is either `1` or `0`.

As we've seen in our earlier examples, the new Classification XML processing has considerable flexibility. For example, for SMF processing we can:

- Add the `SMF_request_activity_enabled` tag to the outer-most protocol section allows you to turn SMF recording on or off for *all* requests for that protocol.
- You could turn it on for all requests for a protocol by coding the new tag on the outer-most protocol node, and turn it *off* for some particular application by coding the tag and a value of `0` on an inner protocol node.
- Or you can turn it on for just one application.
- Or you could turn off SMF recording for Internal work to eliminate 'noise' from work that is just internal to the server.

Here is a simplified example that turns on SMF recording for *all* HTTP requests:

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" SMF_request_activity_enabled="1">
      <http_classification_info transaction_class="Q" uri="/gcs/admin"/>
      <http_classification_info transaction_class="R" uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

Here is an example that turns SMF off for the specific `/gcs/admin` URI:

```
<InboundClassification type="http"
           schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
           host="host.company.com" SMF_request_activity_enabled="1"/>
   <http_classification_info transaction_class="Q" uri="/gcs/admin"
           SMF_request_activity_enabled="0"/>
   <http_classification_info transaction_class="R" uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

There are additional XML tags to control whether the optional sections are written for a particular request. The tags are:

```
SMF_request_activity_CPU_detail     0 | 1
SMF_request_activity_timestamps     0 | 1
SMF_request_activity_security       0 | 1
```

Once again we have those pesky `MODIFY` commands. They allow the SMF recording to be turned on or off dynamically for a server. If you have specific tags in the XML file indicating that SMF recording is on or off for particular requests and you issue the `MODIFY` command to turn recording on (or off), what happens? Just like with the DPM, the `MODIFY` command overrides what is in the XML file. So using `MODIFY` to turn off SMF recording turns it off, no matter what is in the XML file.

How do you get SMF processing to resume honoring the XML file contents? Once again, it is the `RESET` option.

```
F <server>,SMF,REQUEST,RESET
```

This command will tell the server's SMF processing to go back to using whatever configuration is in the classification XML file. Similarly, the `MODIFY` commands for the optional sections will override whatever is in the XML file until you use `RESET` (instead of ON or OFF) to go back to using the XML file contents. Remember that you can use the `MODIFY` command `DISPLAY,SMF` to see what is currently configured.

Message Tagging

WebSphere issues a lot of messages. Some of them are WTOs to the console, some of them are WTOs to the joblog, some of them get written to the error log logstream or `SYSOUT DD`. The server might write trace entries. Applications issue messages or sometimes just write things to standard-out. With multiple applications installed in a single server, or with large applications consisting of multiple parts, sometimes you wonder which thing was responsible for some message. Did that serious looking error message come out because of a problem with an application that is critical to the business or can you go get a cup of coffee and look at it a bit later. It can be hard to tell.

To help with this problem we introduced Message Tagging in Version 8. Simply put, you use the classification XML to associate a string of yours with a request. If WAS writes a message from a thread that we know is processing on behalf of that request we will include your message tag in the message.

We should look at an example. Suppose we update our XML fragment to identify any request that comes under the URI `/gcs/*` with the message tag "GCS". The updates would look like this:

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N"
          host="host.company.com" dispatch_time="300"
          stalled_thread_dump_action="TRACEBACK"
          cputimeused_limit="30 cputimeused_dump_action="TRACEBACK"
          request_timeout="5" timeout_recovery="servant">
      <http_classification_info uri="/gcs/*" message_tag="GCS">
         <http_classification_info transaction_class="Q"
                uri="/gcs/admin"
                dispatch_timeout="1800" queue_timeout_percent="50"
                stalled_thread_dump_action="SVCDUMP" cputimeused_limit="300"
                request_timeout="30"
                dpm_interval=”1000” dpm_dump_action="TRACEBACK"/>
         <http_classification_info transaction_class="R"
                uri=“/gcs/admin/1*” timeout_recovery="session"/>
      </http_classification_info>
   </http_classification_info>
</InboundClassification>
```

Some notes on the updates:

- We added a new level of classification around the `/gcs/admin` and `/gcs/admin/1*` classifications we had all along. This one uses `/gcs/*` as the URI to cover any URI under this application.
- We inherit the transaction class from the parent node ('N') and allow the nodes underneath to specify things like timeouts.
- The message tag of "GCS" will apply to any message issued by a thread which WebSphere can tell is processing a request under this `/gcs/*` URI.

This tag will show up on any messages, traces, or application `println()` commands issued while dispatching a request in this application. Here is an example header of a trace entry and an entry in the error log (`SYSOUT`).

```
Trace: 2011/03/21 22:15:48.298 02 t=6BEE88 c=0.6 key=S2 tag=GCS (0401D00A)
```

```
BossLog: { 0233} 2011/03/24 14:05:52.951 03 SYSTEM=SY1 CELL=WAS00 NODE=NDN1
CLUSTER=BBOC001 SERVER=BBOS001  PID=0X010063 TID=0X3156630000000043 t=6C6938
c=UNK ./bbgrjtr.cpp+717  tag=GCS ... BBOO0220E: SECJ6237E: Authorization failed.
The SAF user MSTONE1 does not have READ access to any of the following SAF ...
```

For a WTO the `tag=GCS` would be added at the end of the message.

WTOs are often used for automation. With this new function it is possible existing system automation might be affected by the presence of the tag at the end of the message. Rather than scrub all the message tags out of your XML file until you can change the automation to tolerate the tag, you can just tell the server not to honor the tags. Set environment variable `ras_tag_wto_messages` to zero (0) to disable the appending of the message tag onto WTOs.

Trace Filtering

Another feature of the new WAS z/OS V8 granular control function is the ability to enable tracing only for certain requests that arrive in a server. First we'll offer a little background on why this new feature is present, then we'll explain how it's used.

Background

WebSphere has an assortment of ways to collect documentation when an error occurs. These include SVCDUMPs, Java dumps, WebSphere First-Failure-Data-Capture, etc. But some problems are just much easier to debug by turning on bits of code buried in the product that produce traces as the code runs. On z/OS these traces can be sent to a buffer and written to a CTRACE external writer if you don't want to worry about them piling up in your JES Spool. Or you can redirect the JCL DD card to send the output to a file, probably in your USS File System.

That is all just wonderful when the volume of running work is relatively low and you only have one thing running in the server. Turning on, for example, the web container trace does not produce too much output if you only run one or two requests while the tracing is enabled.

However, in a real production server, especially one with multiple applications installed, large volumes of different requests can run while you try to capture the trace output for the problem request that you (or IBM) are trying to debug. Even if you can find the room to keep several million lines of output, and even if you can manage to send it all to IBM, you know it will probably take us a while to find what we need in all that output.

New function in WAS V8

Wouldn't it be nice to be able to tell the server you just want tracing for a specific request? Well, you can now give us all these other attributes for a specific request, why not “Trace On”? Good idea. Here's how it works...

The first thing to do is figure out what particular request you want tracing turned on for. You have all the granularity available in the classification XML file. So depending on how much you know about the 'problem request' you might be able to be very specific or you might have to just target a particular application by specifying its context root. We will go back to our original example XML and tag the one specific URI with the tag to indicate we want request-level tracing for HTTP requests that exactly match that URI.

```
<InboundClassification type="http"
          schema_version="1.0" default_transaction_class="M">
   <http_classification_info transaction_class="N" host="host.company.com">
      <http_classification_info transaction_class="Q"
         uri="/gcs/admin" classification_only_trace="1"/>
      <http_classification_info transaction_class="R"
         uri="/gcs/admin/1*"/>
   </http_classification_info>
</InboundClassification>
```

The second thing to do is to activate this XML file. For now we will just assume you configured the server to point to this XML file and restarted the server. You can also update the XML file dynamically as we will see a little later in this paper and that is probably what you would really do to debug a problem in a production environment.

When the server sees that the XML file contains a `classification_only_trace` tag set to `1`, the server goes into a special trace mode. Only traces written from a thread that is dispatching a request that matched a clause in the XML with `classification_only_trace` set to `1` will actually get written.

Unless you normally run with some tracing enabled, the chances are good no trace specifications are set. So you need to turn on some tracing. Normally you would refer to the

PMR where Level 2 gave you instructions about what tracing to turn on. For this example we'll assume the web container tracing is set on since our XML examples are for an HTTP request.

Note: Trace settings may be established dynamically (not requiring server restart) either through a `MODIFY` command or through the Administrative Console.

And that is all there is to it ... almost. Tracing is on for the web container, *but traces will only be produced if they are generated from a thread that is dispatching a request that matched an enabling entry in the classification XML file.* If you turn other tracing on or off you will still only get that tracing for requests that match enabling entries in the XML file.

In other words, rather than web container tracing for all requests for all applications in the server, this limits the tracing to only those requests that match the enabling tag in the XML.

Once you have captured the trace you need, you can turn it off. *Be careful to do this in the right order.* If tracing is enabled and you revert to a Classification XML file that has *no* `classification_only_trace` setting, then tracing will get written for *every request*. So you want to turn off tracing first. Then go update your XML file and remove the classification_only_trace entries (or set them to 0).

What if you make a mistake and trace starts spewing out and filling up spool? Here's a handy modify command to have ready:

```
F <server>,TRACERECORD,OFF
```

This `MODIFY` command turns off tracing. It does not matter what is in the XML file or what tracing you have configured. This just pulls the big red switch and turns it off. Then you can take your time to get the XML file straightened out and activated. When you are ready to let the server go back to honoring the XML file for tracing, you can issue:

```
F <server>,TRACERECORD,RESET
```

Like the other `MODIFY` commands we have seen, `RESET` tells it to go back to whatever the XML file says to do. There is a third flavor of this modify command, namely:

```
F <server>,TRACERECORD,ON
```

This command tells the server to ignore the XML file and just allow whatever tracing is configured to come out.

The `MODIFY` command `DISPLAY,TRACERECORD` will show you the current setting.

Updating the Classification XML File Dynamically

We have mentioned a few times that it is possible to update the Classification XML file without restarting the server. This was not true in WAS releases prior to Version 8. Because we added so much more information into the file we thought it would be very valuable to be able to update the contents dynamically.

Of course you can actually update the file itself any time you want. Just edit the file and change it. The tricky part is getting WebSphere to notice that you changed the file and update its behavior based on the new file content. This is because the server does not re-read the file for each individual request. The contents of the XML file are read at server startup and a classification tree is built internally to the server and used to classify each request.

What we needed was a way to get the server to read the file again and build a new tree.

There are now two ways to do this. The simplest method is to issue the following modify command:

```
F <server>,RECLASSIFY
```

This command tells the server to re-read the configured `wlm_classification_file` and build a new internal tree. If the XML in the file fails to properly parse, then a message is issued and no change occurs.

This is a handy way to activate an update if you are *sure* you got the update right.

A safer way to make an update is to create a *new* file that is a copy of the old one and make your updates in the new copy. Then tell the server to go read the *new* file. If the new file parses successfully but causes other problems you can quickly issue another `MODIFY` command to switch back to the original file without having to try to quickly edit and fix your changes.

The command to tell the server to go read a new file is:

```
F <server>,RECLASSIFY,FILE=path to XML file
```

The nice thing about always using the basic `RECLASSIFY` command (without a `FILE=` parameter) is that you always know what file the server is using -- it will be whatever file is pointed to with the `wlm_classification_file` environment variable.

If you issue a series of `RECLASSIFY` commands with different `FILE=` values, how can you be sure what file is currently in effect? Just issue the command:

```
F <server>,DISPLAY,WORK,CLINFO
```

The server will provide you with a bunch of counters to show how the contents of the file are being used. At the top of the output you will see message `BBOJ0129I`. Here is a sample:

```
BBOJ0129I: The /wasv8/classification.xml workload classification file was loaded
at 2011/11/12 19:33:35.297 (GMT).
```

Which conveniently tells you the full path to the XML file and the time when it was read and the internal tree built. As long as the last-updated timestamp on the file itself is before the load time indicated here, then the contents of the file match what the server is using.

(I'm writing this during what I hope is the final edit of this paper. I'm hoping Don Bagwell won't read through the whole paper again, so I'm going to add in this little comment thanking him for all his help in getting this paper cleaned up and restructured. Thanks Don! With a little luck he won't find out this is in here until its been published.)

Is My Classification File Working?

We mentioned above that the `DISPLAY,WORK,CLINFO` command will show you how the XML file is being searched and used. But it will not show you how a particular request gets classified and what attributes it ends up having. If you are concerned that a particular request might not be getting the right timeout attribute or other characteristics, check SMF.

In Version 8 we made a number of updates to the SMF Type 120 Subtype 9 record. That's a subject for a different paper, but I wanted to point out a couple of relevant changes here. If you look in the z/OS Request Information Section you will find some new fields and some new flags.

The new flags include `SM1209FK` which indicates if classification_only_trace is turned on or off for this request. The flags `SM1209FM`, `SM1209FN`, `SM1209FO`, and `SM1209FP` correspond to the four SMF flags controlling the record and the optional sections. Why do we include those when the presence of the SMF record itself would seem to indicate the record is on? Remember there are `MODIFY` commands that can override the flags in the XML file. The flags in the SMF record tell you what we got from the XML file. That might not correspond to what actually happened at runtime depending on what `MODIFY` commands had been issued.

The various dump actions (timeout, CPU Timeout, and DPM) are encoded in `SM1209FR`, `SM1209FS`, and `SM1209FT`. The timeout recovery option is revealed in `SM1209FU`. The dispatch timeout value is in `SM1209FV`. The queue timeout value is in `SM1209FW`. Note that this is not the percentage value you put in the XML file, but the actual timeout value calculated using that percentage.

The outbound request timeout value used is in `SM1209FX`. The CPU time limit is in `SM1209FY`. The configured DPM interval is shown in `SM1209FZ`.

And, finally, any message tag you configured will also show up in `SM1209GA`. This tag might be a very handy way to sort your SMF records by application or whatever you choose to use to group related requests.

Document change history

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

11/14/2011	Original Version
11/21/2011	Added document number
10/7/2013	Correct variable names for queue timeout percentages (Thanks Raymond)
12/3/2013	Correct typos in cputimeused_limit section (Thanks to the customer who told Evan)

End of WP102023