Many thanks to Don Bagwell, Mike Loos, Mike Cox, Mike Ginnick, Adam Wisniewski, Tim Spewak, Ed Mezarina, Chris Gianfrancesco, and probably a bunch of other people I forgot..
# Table of Contents

**Overview**
- Functions in 7.0.0.29 released on June 24, 2013.................................................................4
- Functions in 7.0.0.31 released on December 19, 2013.........................................................4
- Functions in 8.0.0.6 released on April 29, 2013.................................................................4
- Functions in 8.0.0.7 released on August 19, 2013..............................................................4
- Functions in 8.0.0.8 released on December 16, 2013.......................................................4
- Functions in 8.5.0.2 released on April 15, 2013...................................................................5
- Functions in 8.5.5 released on June 14, 2013.....................................................................5
- Functions in 8.5.5.1 released on November 11, 2013.........................................................5
- Functions in 8.5.5.2 released on April 28, 2014.................................................................5

**Functional Detail**
- PM74923 – Better Living Through Server Output Management........................................6
- PM87193 – Handling more than 256 servers on one LPAR..................................................6
- PM82634 – Work with affinity and timeout_delay.................................................................7
- Display Work Support for Messages from the CRA..............................................................9
- Display Work ‘Highwater’ Reporting...................................................................................10
- Use of the WLM Health API..............................................................................................11
- Tracing and Logging Large Callstacks and **BUFFER OVERFLOW**................................12
- Dynamic Controller Thread Pool Management................................................................13
- Better Wildcarding in your Classification XML file............................................................14
- What is THAT task doing? Trace filtering by TCB...............................................................14
- Easier Modify Command Automation – CART support at last............................................15
- PM93967 – Improvements to MSGROUTE......................................................................15

**Document Change History**
.........................................................................................................................17
Overview

In the previous 'Hidden Gems' papers I've focused on little known features 'hidden' inside specific releases: WP101138 covered Version 6.1 (and previous releases), WP101464 covered Version 7, and WP101992 covered Version 8. But those papers start by covering 'gems' that have also shown up in the service stream. We've had a number of interesting items turn up in service in the last year that I thought deserved some special attention, and perhaps there might be some more coming along. With that in mind I thought perhaps it was time to create one 'hidden gems' paper that just covered things in service.

First let's go through the service levels we are covering and the contents we'll be talking about. Then either take the link to the topic you're interested in, or just keep turning pages and read them all!

V7

Functions in 7.0.0.29 released on June 24, 2013

- PM74923 – Better Living Through Server Output Management

Functions in 7.0.0.31 released on December 19, 2013

- PM87193 – Handling more than 256 servers on one LPAR
- Display Work Support for Messages from the CRA
- Display Work 'Highwater' Reporting
- Use of the WLM Health API
- Tracing and Logging Large Callstacks and **BUFFER OVERFLOW**

V8

Functions in 8.0.0.6 released on April 29, 2013

- PM74923 – Better Living Through Server Output Management

Functions in 8.0.0.7 released on August 19, 2013

- PM87193 – Handling more than 256 servers on one LPAR
- PM82634 – Work with affinity and timeout_delay

Functions in 8.0.0.8 released on December 16,2013

- Display Work Support for Messages from the CRA
- Display Work 'Highwater' Reporting
- Use of the WLM Health API
- Tracing and Logging Large Callstacks and **BUFFER OVERFLOW**
- Dynamic Controller Thread Pool Management
- PM93967 – Improvements to MSGROUTE
V8.5

Functions in 8.5.0.2 released on April 15, 2013
PM74923 – Better Living Through Server Output Management

Functions in 8.5.5 released on June 14, 2013
PM82634 – Work with affinity and timeout_delay

Functions in 8.5.5.1 released on November 11, 2013
PM87193 – Handling more than 256 servers on one LPAR
Display Work Support for Messages from the CRA
Display Work 'Highwater' Reporting
Use of the WLM Health API
Tracing and Logging Large Callstacks and **BUFFER OVERFLOW**

Functions in 8.5.5.2 released on April 28, 2014
Dynamic Controller Thread Pool Management
Better Wildcarding in your Classification XML file
What is THAT task doing? Trace filtering by TCB
Easier Modify Command Automation – CART support at last
PM93967 – Improvements to MSGROUTE
Functional Detail

In the sections that follow we'll go through each of the various new functions mentioned above...

**PM74923 – Better Living Through Server Output Management**

Introduced in: 7.0.0.29, 8.0.0.6, and 8.5.0.2

This one is described in great detail in IBM Techdoc WP102267 which can be found here: [http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WeblIndex/WP102267](http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WeblIndex/WP102267)

I wanted to mention it in this document because its a pretty cool thing (cool enough to get a techdoc of its own). Put simply, its a better way to make server output available to developers on z/OS. By default the server output goes to the job SYSPRINT and SYSOUT DD cards which are usually just sent to SYSPUT=* and end up in the job output. Developers generally don't want to log onto TSO and deal with SDSF etc. to find where their application println's went.

Over the years people have taken various approaches to dealing with this, all discussed in the techdoc above. This new option makes it possible to get the output sent to a file in the USS file system (where the developers will be much happier looking for it) and into a uniquely named set of files. You can also force it to roll to a new set of files via a MODIFY command (probably driven by automation). Messages are written at the bottom of the old log and top of the new log to help you (and IBM Level 2!) put them in the proper order.

Lots more information in the WP102267 techdoc...including a cool way to set up your IHS Apache Web Server to provide easy read-only access to the logs. Go check it out...

**PM87193 – Handling more than 256 servers on one LPAR**

Introduced in: 7.0.0.31, 8.0.0.7, and 8.5.5.1

As more and more customers use WebSphere Application Server on z/OS and as existing customers make more and more use of it, we run into some interesting limitations. In this case it turns out that there is an interesting restriction in the z/OS Workload Manager as regards WAS Controller Regions.

To explain we need to back up a bit and understand how IIOP requests for remoteable EJBs are handled. When a client application wants to use RMI-IIOP to access an EJB located in a WAS on z/OS server it first needs an object reference called an IOR. The 'ORB' code used by the client application can work with this IOR to find and send the RMI request to the appropriate server using IIOP flows (you can search the web for those acronyms if you like, but for the most part just accept that the client gets a reference to the target object and it gets resolved magically).

When the server is on z/OS, the Daemon gets involved. The Daemon knows about all the servers from his cell that are active in the sysplex. When asked, the Daemon can help the client find the right server. It does this with some help from WLM.

WLM knows about all the WAS servers that are present on each system in the sysplex and can help the Daemon find the right one to which the request should be routed. How does it know which servers are up? Because the servers call a WLM API (IWMSRCRR) as part of their initialization process.
This is where it gets interesting (if it wasn't interesting for you already). It turns out that WLM has a restriction that only 256 servers per LPAR can register with WLM using this API. Must be a fixed size array somewhere or something...I don't know.

So what happens if you try to start 257 WAS servers on the same LPAR? The last one up will issue message BBOO0037E reporting a return code of 8 and a reason code of 00000868 from IWMSRCRR. Nothing else obviously bad happens. The server still comes up. However, the Daemon will never be told by WLM to send an IIOP request to servers whose IWMSRCRR call failed.

Do you care? Maybe not. If your applications don't have any remoteable EJBs then you may not care. If you aren't using IIOP for admin flows then you may not care. But its likely that you care a little bit. If you have this many servers on one LPAR odds are some of them probably have some EJBs. In this case it may be important that some of the servers successfully complete their IWMSRCRR calls. For other servers you may not care.

How do you manage this? Well, if you don't care at all or if you are well under 256 servers per LPAR there's nothing to manage. But if it matters then you can try to arrange that the servers that have to register get started first. If you get above 256 servers later on it won't matter as long as those servers don't get involved with EJB routing. But that's a pretty messy solution and if one of those EJB servers gets restarted it still might fail.

You need a way to tell WebSphere which servers have to be able to register successfully and which ones you don't care about. So with PM87193 we introduced a new configuration environment variable called protocol_iiop_enable_wlm_routing. This variable, which defaults to '1', indicates whether this server needs to call the IWMSRCRR service or not. If it does (1), the server uses the API. If not (0) then it doesn't.

So, if you have 256 servers on one LPAR (or you're getting close) you might look to see which servers really need EJB routing support and which ones don't and consider setting this new variable to '0' for the servers that don't. Once you're at the maintenance level that supports it of course.

### PM82634 – Work with affinity and timeout_delay

Introduced in: 8.0.0.7 and 8.5.5

Ever try really hard to do the right thing and get in trouble anyway? Happens to me all the time. Happened to us in WAS Version 8. We tried to improve how requests were handled during timeout processing and wound up causing problems. We were so proud of what we'd done that I even wrote it up in the V8 Hidden Gems paper (WP101992). To explain, I'll start by just including the text from that paper that talks about what we did, then I'll be back to explain how that got us in trouble and what we've done about it.

---Start of text from WP101992---

When timeout processing decides it has to abend a servant region there might be other requests in dispatch on other threads in that servant that are completely innocent. For that reason you can configure a timeout delay (control_region_timeout_delay or use the Modify TIMEOUT_DELAY command). This defers the abending of the servant region for the specified number of seconds. During that delay the servant region dispatch threads will finish what they are doing and not take any new work. This allows the innocent to get out of the way.

But there is a complication. What about requests that come in during the delay that have affinity to the servant region we are going to abend? Well, they just sit in the WLM queue. The servant region dispatch threads will not pick up new work, but the servant is still a valid target for the request. These requests will remain in the queue until their queue timeout expires or, more likely, until the timeout delay runs out and we abandon the servant. The abending of the servant region will
either cause the requests to be cleaned up (rejected) or requeued to be able to run in any servant region.

This requeue of the request is essentially the same as what would happen to the request if it came in just after the servant region was abended. We can not find the servant region it has an affinity to, so the request is allowed to run in any servant. The state data that caused the affinity should be tucked away in DRS and able to be recreated in any servant.

Well then, why do we let these requests sit in the queue waiting for the abend? Why not just queue them to run anywhere right away? Because the request actually has an affinity to an active (but soon to die) servant region. Allowing it run in a different servant region when the original one is not yet dead would cause a lot of confusion and problems.

But letting the request just sit in the queue is bad. Somewhere there is a client waiting for a response. Should we just wait for the timeout delay to expire and then requeue it when it is safe? What if you only have one servant region? In that case the request will have to wait for the timeout delay to expire AND for a new servant region to initialize. That might take a while. Perhaps it would be better to just reject the request.

So in V8 we changed the code a little bit and that is what happens. Any request that comes in with an affinity to a servant region that is going to be abended when the timeout delay expires will just be rejected. If the client re-sends the request a little later, after the servant has abended, then the request can run in any servant and will be queued without affinity just like before. Note that any requests with affinity that are actually in the queue to run when the servant is marked for an abend will also be rejected.

This should keep clients with requests with affinity to a dying servant region from hanging during the timeout delay. Remember that we have the timeout delay to avoid impacting requests that were not responsible for the timeout that led to the abend. This helps us not hurt newly arrived innocent requests too badly either.

----End of text from WP101992----

Ok, we're back. That sounded pretty good, right? Well, we had a customer who was configured with two servant regions. All the requests were running in the first servant region. The users of the application were on the phone with real customers, asking questions, and using the application to populate state data from the customer's answers. If a timeout occurred in that servant region (prior to V8) the timeout delay allowed requests in flight to finish, new requests with affinity waited in the queue as described above. When the timeout delay expired (15 seconds for this customer) the affinity requests got requeued to the other servant region, picked up their state data from DRS and continued on. To the application users (and more importantly the customers they were talking to) this appeared as a 15 second 'hang' which we recovered from.

Now comes V8 and the same situation occurs. In this case we just reject the affinity request that comes in during the timeout delay. The user of the application is forced to begin again creating new state data from scratch. Which meant starting all over with the customer asking all the same questions again. Not very popular.

So it turns out that sometimes you might want to immediately reject a request that isn't going to be able to run for a while (because its holding up important things up-stream) and sometimes you just want to wait it out and hope things clear up. It depends on the application and how it is used.

And thus we come to PM82634. In this APAR we let you decide how you want it to behave. There is now yet another environment variable, this one called control_region_http_requeue_enabled. When set to zero (the default) you get the 'immediate failure' behavior as we changed it to in V8. But if you set it to one, then you're mostly back to the original behavior. The original (pre-V8) code just queued everything with affinity to a timed out servant and let it get cleaned up when the servant died. In V8 we
Hidden Gems in the WAS z/OS Maintenance Stream

changed to not queue anything with affinity to the dying servant during the timeout delay. And with PM82634 and control_region_http_requeue_enabled=1 we look at the request with affinity. If it is a request we are able to requeue to another servant we will go back to the original behavior and let it wait on the queue for the timeout delay to expire, the servant to abend, and then get requeued to another servant. However, requests with affinity to the dying servant that can't be requeued (there are cases like this) will still be immediately rejected. These requests are going to fail anyway because they can't be requeued so there is no reason to wait.

Hopefully this time we did some good.

Display Work Support for Messages from the CRA

Introduced in: 7.0.0.31, 8.0.0.8, 8.5.5.1

Way back in WebSphere Version 5 we introduced the DISPLAY,WORK command (underneath the MVS 'Modify' command for the WAS controller). This pretty cool command lets you see how much work had been processed by the server since it started and how much work was actually in the server at the time. You could even look server region by server region and see how work was spreading (or not) across them. Of course the data is just a point in time. For a server running pretty hard the numbers are obsolete by the time the messages make it to the console.

But for automation purposes (or just to give you a good vibe that stuff is happening) its not a bad way to monitor and make sure work is still flowing. In fact, the output of the command even shows you the difference (delta) in the 'total requests ever' value since the last time you issued the display command. This made it easy for automation to monitor progress. If the delta is ever zero, then nothing happened in the server since the last command. Depending on what you're monitoring, that might be interesting.

I could spend more time here talking about the different values returned for the different types of work and the other options available under the display work command. But that's not the point of this section.

What is the point? Well, first we have to go back in time a bit. Back in Version 6, we created an optional additional address space called the CRA or Controller Region Adjunct. Its really more of a special servant region, but that's its name. The CRA existed to host the integral messaging engine (sometimes called the SIBus). Messaging work that came from the SIBus flowed into the controller and through some special paths there into the normal servant regions where it ran kind of like other work. This code was all pretty special case stuff and didn't get wired into the code that does all the counting to support display work.

Then in Version 7 the MQ folks came along and gave us Activation Specification support to get messages from MQ. That support got placed into the CRA also. And the work flow went through the same paths in the controller into the servant region.

At first not many people used either SIBus or Activation Specifications and it didn't really matter that the display work command wasn't wired in. We talked about adding it, but all the hooks are in the controller and the controller code doesn't know the difference between SIBus work and Activation Spec work flowing through it. Same path. The controller is really just a pipeline between the CRA and the SRs. Making the display command tell the difference was going to be complicated.

Over time we got more and more requests to have display work support things coming out of the CRA. We also realized that customers were either going to be SIBus or Activation Specs and were unlikely to be using both in one server. So maybe it didn't really matter why it came out of the CRA, just that it did.
Having made it over that hurdle, the rest was actually pretty easy. So, in the maintenance levels listed above, we've added a new keyword to the DISPLAY,WORK command: CRA. Here's an example:

F BBOS001,DISPLAY,WORK,CRA
BBO00255I TIME OF LAST WORK DISPLAY 2013/04/30 21:31:37.470352
BBO00256I TOTAL CRA REQUESTS 0 (DELTA 0)
BBO00257I CURRENT CRA REQUESTS 0
BBO00258I CRA REQUESTS IN DISPATCH 0
BBO00410I HIGHWATER CRA REQUESTS 0
BBO00267I TOTAL CRA TIMEOUTS 0 (DELTA 0)
BBO00188I END OF OUTPUT FOR COMMAND DISPLAY,WORK,CRA

And the SUMMARY and ALL command outputs have been modified to now include work coming from the CRA.

**Display Work 'Highwater' Reporting**

Introduced in: 7.0.0.31, 8.0.0.8, 8.5.5.1

One of the values you can get from the various DISPLAY,WORK commands is a count of the current number of requests in the server (or servant). This is pretty cool because by monitoring it you can get a rough idea of how much is going on usually and maybe spot a sudden surge (or drop off) that automation might be able to react to.

In various conversations with customers who use the display work output, we got suggestions to add a 'highwater' counter to the display output. Basically, while its all well and good to know how many requests are in the server right now, it might also be useful from a capacity planning perspective to know the maximum value seen in the life of this server.

For a controller this value tells you how far behind this server ever got (if it did). So if the current value for work in the server is 150, but the highwater value is 500, then perhaps 150 isn't too bad. But if that IS the highwater value, then maybe its a spike you should care about.

So while we were in all this code for the CRA work counting described above, we also added a bit more to track the highwater value for 'current work' for each of the work types. If you look closely, you'll notice the new line (message BBOO0410I) in the CRA display output above. But in case you prefer a more classic flavor of the command, maybe HTTP requests, here's another sample:

F BBOS001,DISPLAY,WORK,SERVLET
BBO00255I TIME OF LAST WORK DISPLAY 2013/05/13 21:04:27.773915
BBO00256I TOTAL SERVLET REQUESTS 0 (DELTA 0)
BBO00257I CURRENT SERVLET REQUESTS 0
BBO00410I HIGHWATER SERVLET REQUESTS 0
BBO00258I SERVLET REQUESTS IN DISPATCH 0
BBO00267I TOTAL SERVLET TIMEOUTS 0 (DELTA 0)
BBO00188I END OF OUTPUT FOR COMMAND DISPLAY,WORK,SERVLET

The next question I usually get is why HTTP requests are listed as 'servlet' when they could be JSP's or other things. Does this count only include things that are actually servlets? The answer is that anything that comes in over HTTP ends up counted as a 'servlet'. When the support went in back in V5 we thought we'd eventually separate out the counters by type of work rather than by how we got it. We went with 'servlet' at the time planning to sort it all out.
Hidden Gems in the WAS z/OS Maintenance Stream

later. But things didn't work out the way we thought they would and the code doing the counting doesn't really know what you're going to run with the request, just how we got it. And because automation exists that issue these commands and parse the output, we can't just change them. I suppose we could introduce a new flavor of display work (DISPLAY, WORK, HTTP maybe?) that did exactly the same thing as DISPLAY, WORK, SERVLET. But does that help or just make it even more confusing? Now you'll wonder what the difference is between HTTP and SERVLET.... best to leave it as it is.

Use of the WLM Health API

Introduced in: 7.0.0.31, 8.0.0.8, 8.5.5.1

This one came at us from two directions. First of all, z/OS WLM provided an API a while ago that allows a server to indicate its 'health' to WLM. Health is represented as a percentage. So a server that is fully functional and all good should be 100% healthy. Probably when initialization completes you would say a server is ready to go and so is 100%. Maybe. But when do you tell WLM the server is less healthy? And when does it come back to 100%? How does it know? We had a lot of internal discussions and quite a few interesting ideas, but nothing ever came of it.

But then we had a conversation with one of our larger customers who runs a large set of clustered servers handling a pretty significant traffic volume. They feed these servers HTTP requests routed through Sysplex Distributor. When they recycle a server in the cluster and it finishes initializing Sysplex Distributor sees the HTTP ports open and starts sending its share of the incoming work. However, there are server and application caches which are not populated yet because the server has done no application work yet. The JIT has not yet done any serious compilation of application classes since it has not yet observed any application requests running. In short, for a while, the server gets his fair share of requests but doesn't do as good a job handling them as the other cluster members who have been up longer. It would be nice if we could scale his workload up over time as he 'warmed up'.

Eventually these two thoughts came together and resulted in this enhancement. We need to go back to that idea that the server is 100% healthy after initialization is complete. Perhaps it really isn't. The idea we had was to start the server off as pretty unhealthy and then increase its health over time until it reached 100%. This would cause routers like Sysplex Distributor to gradually ramp up the server's workload over time as it warmed up.

To help you do this we have provided two new environment variables: wlm_health_increment and wlm_health_interval. These work together to control how we increase the health value over time. Basically we start at a health value of zero and increase it by the 'increment' every 'interval' seconds until we get to 100. Every time we call WLM you will see a BBOO0411I message telling you the new health percentage. Note that the initial message at zero percent is issued during initialization and depending how long it takes to get fully up will affect when the first non-zero percentage comes out. They should be separated by (roughly) your configured interval after that. Here's an example set of messages:

```
wlm_health_increment=15
wlm_health_interval=15

11.56.14 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 0
11.56.44 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 15
11.56.59 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 30
11.57.14 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 45
11.57.29 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 60
11.57.44 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 75
11.58.00 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 90
11.58.15 STC00085 BBOO0411I SERVER WLM HEALTH PERCENTAGE IS NOW 100
```
Hidden Gems in the WAS z/OS Maintenance Stream

**Tracing and Logging Large Callstacks and **BUFFER OVERFLOW**

Introduced in: 7.0.0.31, 8.0.0.8, 8.5.5.1

Have you ever been paging through your SYSOUT DD output looking at various messages that might relate to whatever problem you're having and run across a long message, probably a call stack, that ends with ***BUFFER OVERFLOW***? That sounds pretty serious. Before we get into what we've changed here, let me explain what's going on.

Whenever a message gets logged from code running in Java in WebSphere it gets routed to some native code that actually writes it to SYSOUT. That code has to convert the text into EBCDIC so you can read it in SYSOUT. The resulting message has to fit in one buffer. We could have figured out how big a buffer we needed and just allocated storage and moved on. But we were worried (especially back in the 31-bit days) about some giant message (probably a really long callstack or something) coming through and running us out of native heap storage trying to convert it to EBCDIC. So we limited the amount we would convert to what we thought was 'reasonable' and just tacked on the rather unfortunate string ***BUFFER OVERFLOW*** to indicate we'd run out of room and just stopped converting.

This overflow situation also has a tendency, in messages consisting of a large callstack, to cut it off just as you get to the part you were really interested in. Honestly that's not by design. It just somehow always seems to work out that way.

In this update we've kind of weasled our way around this. Whenever we get into this situation, we report back up to Java that we ran out of room and let the Java code call the FFDC code to create an FFDC entry that will have the whole callstack.

In the SYSOUT (or SYSPRINT as shown here) you'll see something like this:

```
Trace: 2013/05/29 16:36:41.435 02 t=6C9140 c=UNK key=P8 tag= (13007004)
SourceId: com.ibm.ws.tx.jta.RecoveryManager.filterClassPaths
ExtendedMessage: Entry; /webV80I/1pp/zWebSphere/V8R0/java64/J6.0.1_64/lib:/we ar:/webV80I/1pp/zWebSphere/V8R0/java64/J6.0.1_64/lib/annotation.jar:/webV80I/1pp
V80I/1pp/zWebSphere/V8R0/java64/J6.0.1_64/lib/charsets.jar:/webV80I/1pp/zWebSphe
rbSphere/V8R0/java64/J6.0.1_64/lib/ibmcertpathfw.jar:/webV80I/1pp/zWebSphere/V8R

....big giant callstack here....
```

**server/properties:/WebSphere/ND/AppServer/****BUFFER OVERFLOW***TRACE DATA ROUTED TO FFDC REPORTING A NativeMessageOverflowException***

Note that the 'buffer overflow' now indicates that the data is in an FFDC. You should also get information from FFDC that it wrote a record, like this:

```
Trace: 2013/05/29 16:36:41.439 02 t=6C9140 c=UNK key=P8 tag= (13007004)
SourceId: com.ibm.ws.ffdc.impl.FfdcProvider.logIncident
ExtendedMessage: FFDC1003I: FFDC Incident emitted on /WebSphere/ND/AppServer/ r1_BBOS001S_STC00172_0000018800000001_19bc39d5_13.05.29_16.36.41.437887941282300102
2983.txt com.ibm.ejs.ras.CB390TraceEventListener.
processEvent NativeMessageOverflow_1
```

And the actual FFDC will have the sequence number (here it's '1') as part of the title so you can match them up. So now when you really need those next couple of lines of the callstack, you'll know where to find them.
Dynamic Controller Thread Pool Management

Introduced in: 8.5.5.2

Most of the processing done in the controller region is done in a single pool of threads. Since the very earliest days of WebSphere there were 25 threads in that pool. I'd like to think there was some calculation or reasoning behind that number, and perhaps there is. But regardless it has turned out to be a pretty good number for a lot of years. And actually it remains a good number for almost all of our customers.

However, there are a few special situations. We have occasionally run into situations where the configuration and circumstances are just right so that all of those 25 threads are in the midst of doing some processing but have to wait for some event in order to complete. The only way for that event to occur is to do some processing on one of those 25 threads. But all of those threads are blocked and the server hangs. Its pretty rare. But it has happened. Years ago we introduced a configuration variable (control_region_thread_pool_size) that allows you to change the size of the thread pool. A few customers have run into this situation and had to change the value, but only after the server seized up on them, naturally during the busiest part of their day.

This happens so rarely that there always seemed to be more important things to do with the resources required to address this issue. But we finally found a design for a solution that we were happy with and some time to implement it.

Basically we added some monitoring code in the controller to keep an eye on the queue of work that feeds that thread pool. If we notice the queue isn't moving (e.g. the 'head' element is staying the same) then we might be stuck so we a dynamically add more threads into the pool. We'll keep monitoring and adding threads as we think necessary up to a maximum.

The maximum is important. Firstly because we don't want to just keep creating threads forever. At some point something is seriously wrong. Having a maximum also allows us to indicate to you that something is seriously wrong. When we have added threads and added threads and finally hit the configured maximum, we will issue this message:

BBOO0412I THE MAXIMUM NUMBER OF WORKER THREADS HAVE BEEN CREATED. MAXIMUM=%d

If you see this message from a controller region there is a very good chance the server is hung and will need to be recycled. Or you can raise the maximum.

How do you configure the maximum value? There are two ways. The static configuration uses environment variable control_region_thread_pool_maximum_size. The default value for this variable is zero (0) which tells the server to calculate a maximum value based on the number of servant regions and the number of threads in those servants.

After the server is up and running, you can dynamically change the maximum controller thread value using this modify command:

MODIFY server,WORKERTHREADMAX=

So what should you do? Probably nothing except maybe consider adding some automation to watch for the new message and abend the controller if it happens. Its probably stuck. If it happens more than once you might consider raising the default maximum value.
**Better Wildcarding in your Classification XML file**

Introduced in: 8.5.5.2

In the beginning....well, ok, maybe not that far back. But some days WebSphere Version 5 seems like it was that long ago. Anyway, Version 5 introduced the Classification XML file which you can use to assign transaction class names to application requests to get them classified properly with Workload Manager. (In Version 8 we jazzed that up to let you specify a bunch of other cool stuff too). The original support was somewhat rigid in its support for wildcards. Basically we let you put an asterisk (*) at the end of strings (like a URI) and thus match the start of any string. If you wanted to match on anything else (like *.jpg) you were out of luck.

People complained. So in Version 7 we touched up the code a little bit to let you put that asterisk anywhere you wanted in the string. That let you do things like match `/my/context/root/*.jpg`. But we still restricted you to that single asterisk.

People....gently suggested improvements. And here you go. With the maintenance levels listed above you can put in asterisks to your heart's content. So something like `/my/context/*/mypages.*` will will work.

**What is THAT task doing? Trace filtering by TCB**

Introduced in: 8.5.5.2

I've only been able to think of two scenarios where you might want to use this new feature. The first was the one that led to its creation. We had a thread accumulating CPU in a WAS servant region. Taking occasional javacore's to get the callstack for the thread weren't really helping us figure out what it was doing. Somebody suggested just turning on ALL of the WAS tracing for a few seconds and see what we got from that thread. In an active server that could still yield a huge pile of output and a significant impact to performance. Then Mike Cox said (paraphrasing a bit), “Wouldn't it be cool if we could restrict the active tracing to just this one thread?” Well, now you can.

Direct the TRACEBYTHREAD modify command at the controller region and specify the ASID (in hex) and the thread (TCB address in hex) that you want to trace. Like this:

```
MODIFY server,TRACEBYTHREAD,ASIDX=nnnn,TCBX=nnnnnn
```

Then turn on whatever tracing you want.

When you have what you want, turn the tracing back off (its important to do this first) and THEN disable trace by thread with the RESET command, like this:

```
MODIFY server,TRACEBYTHREAD,RESET
```

You could also use this to capture tracing of application dispatch processing from one thread. Suppose Level 2 wants you to turn on web container tracing in a high volume server with 40 threads per servant and 3 servant regions. Using this feature you could limit the actual tracing to just one dispatch thread in one servant. Of course that thread has to get used to dispatch something, so this only works if you're running the server hard enough to use all those threads. Or you have to be very lucky.
**Easier Modify Command Automation – CART support at last**

Introduced in: 8.5.5.2

Here's one that frankly I'm amazed we haven't been beaten up about yet. When you programmatically issue a console command (like a WAS MODIFY command) you can specify something called a CART on the MGCRE macro. 'CART' stands for Command And Response Token. WTOs issued in response to your command are supposed to be issued with this same CART value. This allows automation to issue 'Display' commands and find and process the resulting WTO responses.

Since the very beginning of Modify command support in WebSphere we have failed to do this. And, to my knowledge anyway, no one has ever complained about it. Over the years we have added more and more commands (mostly under the MODIFY ‘DISPLAY’ support) that are helpful to automation attempting to monitor WAS servers. And as the code that issues the WTOs that are the responses to these commands got scattered through the product it got harder and harder to do anything about it.

Until one day a brave member of our team stepped up and took this one on. Lots and lots of little tiny changes passing the CART around and making sure WTOs got issued properly using it.

There's actually no action required on your part. WTOs issued in response to console commands should now be issued with the proper CART that was provided with the command.

On the slim chance (we hope) that this might actually mess up existing automation, we've provided a way to go back to the original behavior with no CART values on the WTOs. Just set environment variable

```
WTO_with_CART_enabled=0
```

That will put things back the way they've always been.

**PM93967 – Improvements to MSGROUTE**

Introduced in: 8.0.0.8, 8.5.5.2

In Version 8 we introduced a new feature to let you re-route messages issued by WebSphere wherever you wanted. The problem was that the developer of the WAS code got to decide if a message went to the console, or to hardcopy or to the 'error log' (SYSOUT DD). But some customers wanted messages to go other places than where the developer thought it should go. Messages that went to SYSOUT are hard to use for automation, so customers asked for them to be changed to go to hardcopy. Chatty messages to hardcopy were requested to be changed to go to SYSOUT. So in version 8 we gave you a way to change where a message went using one of these environment variables or modify commands:

```
ras_message_routing_errorlog
ras_message_routing_hardcopy
ras_message_routing_console
ras_message_routing_none
```

or

```
MODIFY server,MSGROUTE,ERRORLOG
```
The problem with this great feature is that while the message is now going where you wanted, it isn't going where everyone expects it. So Level 2 might conclude something didn't happen because they didn't find a message they expected in SYSOUT. But the reason it isn't there is because it's been re-routed to hardcopy. What we apparently really wanted was a message routing function that makes a copy of the message. So now we've introduced three new variables and modify commands. These will direct a COPY of the specified message IDs to the indicated destination but leave the original message alone, going to its normal destination. The variables and commands are:

ras_message_routing_copy_errorlog
ras_message_routing_copy_hardcopy
ras_message_routing_copy_console

and

MODIFY server,MSGROUTE,COPYERRORLOG
MODIFY server,MSGROUTE,COPYHARDCOPY
MODIFY server,MSGROUTE,COPYCONSOLE

© 2013, IBM Corporation
Americas Advanced Technical Skills
Version Date: Friday, May 02, 2014
Document Change History

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

November 18, 2013  Original document at time of Techdoc creation
November 19, 2013  Update with document number
November 25, 2013  Fixed typo in Display,Work,CRA output
March 6, 2014     Updates for 7.0.0.31 and 8.0.0.8
May 2, 2014       Updates for 8.5.5.2