WebSphere Application Server for z/OS

What I Can Tell About You
From Your WebSphere SMF
120-9 Records
(or I Know What You Did Last Test Run)

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number WP102311 under the category of "White Papers"

Version Date: October 15, 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
Many thanks go to ...Don Bagwell, John Hutchinson, Mike Loos and the many customers whose data I've looked at.
Introduction

WebSphere Application Server introduced the SMF 120-9 record in Version 7 which became Generally Available in September of 2008. It takes a while for customers to move to a new release and then it takes a while longer to start exploiting the new features. But eventually we started to see PMRs with questions and issues with the data in the SMF 120-9 record.

In Version 8 we added in some new features (affinity tracking, async request data, WOLA outbound request tracking). And we have begun to see customers slowly using these new features.

But something else interesting started to happen. Customers started to send in SMF 120 records as part of the documentation for a PMR about something that wasn't SMF related. Along with the dumps and traces and logs we got huge piles of SMF data. And so I got involved in trying to sort through the data and figure out what it could tell us.

Over time I built up a small toolbox of analysis routines and you can look at WP102312 for an overview of those. But in this paper I wanted to just take those tools and some sample data and go through an analysis.

Unfortunately I can't use real customer data for this, so I asked Don Bagwell to just throw some work at a WAS server and send me the SMF 120-9 records. We'll run some various tools at that data and go through what we can learn from it.

Ready? Here we go....

Thinning Out The Data

The first thing I usually do when first looking at some SMF data is figure out what I've got. The RequestsPerServer report is pretty good at that. It shows you all the systems and servers that wrote SMF records for requests they processed. It also shows what type of requests were processed.

Take a look at the RequestsPerServer data provided with this paper. Like all the data provided, it is in comma-separated-value format. Most spreadsheet software should be able to deal with this format using the commas as delimiters to create columns.

So what do we have? Well, for our simple data it's pretty boring. We have data from only one server. But I do frequently get data for a lot more. One customer sent in SMF data that included activity on 90 different servers. If that's the case, use the MatchServer property (technically com.ibm.ws390.smf.smf1209.MatchServer) to get the browser to only process data from the server you care about. In some of the later reports that will be necessary or you will just drown in data.

You can also use the MatchSystem property to thin the data to just the servers on a particular z/OS image. Use one or the other of these properties depending on what you want to see. If you have a really huge pile of data you might consider using the ReWrite browser plugin in combination with MatchServer or MatchSystem to produce a new SMF dataset that just contains the data you want. This makes all the other reports run faster because the browser doesn't have to process and ignore all the data you don't care about.

What else does this report tell us? Well, we can easily see that Don ran 5000 HTTP requests into his server. And we got one MBean request. This is probably some admin thing that we will want to ignore. We have noticed that with the inclusion of the Virtual Enterprise (or Intelligent Management) into the base WAS product in Version 8.5 that there can be a noticeable amount of background chatter in the form of internal requests and MBeans. If that is the case for the data you have, you might want to use the ExcludeInternal property (com.ibm.ws390.smf.smf1209.ExcludeInternal=TRUE) to have the browser automatically ignore
records for these 'internal' requests. This lets you focus on the application requests that are probably what you are most interested in. Again, if you have a huge amount of these you can use ReWrite with ExcludeInternal to produce a new dataset without them.

If you are running on Version 8 (or up) and have any async beans running you might also see records were written to track those. Recording for async beans is optional and is turned on separate from the 'regular' work records, although both types are recorded using the 120-9 record.

What's in a Name?

If you are looking at your own SMF data, chances are you already know the names of your servers. But if not (or if you are new to this environment), SMF is a good way to figure out naming conventions if you have data from a lot of servers. The RequestsPerServer report shows you the cell, node, cluster, and server names for all the servers that wrote data. It also shows the MVS system name where that server lives. From this data you can probably draw a topology map showing which servers are where and which ones are clustered together.

By spotting patterns in the naming conventions you might be able to notice missing servers. For example, if every cluster seems to have four members spread across four systems, but one cluster only has three members what happened to that cluster member on the system where it didn't show up? Was it down? Is it incorrectly configured to not write data? Or is it up but not getting work for some reason? The absence of data is sometimes as telling as the data you have.

What other names do we have? Let's look at another report. This one is from CSVExport. This report just takes most of the data from each record and turns it into a comma-separated-value row. Pulling that into a spreadsheet gives you 5001 rows (which matches the RequestsPerServer summary).

Scroll across the top of the data and look at the column headings. Early on you will find CRJobname and CRJobid. Later on you will find ServantJobname and ServantJobid. These values will let you find the job output for the controller and servant(s) involved.

A Peek Into Configuration

While we're poking around in the CSVExport output we can glean some interesting things about the configuration.

Maintenance Level and GMT Offset

Actually the first thing to note isn't really configuration, but server maintenance level. Way over on the left in the spreadsheet you will see Release, ReleaseX, ReleaseY, and ReleaseZ. For our data these are 8,0,0, and 7 so we're at 8.0.0.7. Congratulations to Don for being current (as of this writing).

A little farther to the right you will find SystemGMTOffset. This can be used (with some math) to adjust the GMT timestamps we'll look at later to local time. I usually just work in GMT and keep the GMT offset in mind when talking about the data, but if you have to work in local this is how.

The WLM Goal

Scrolling way over to the right (column BK or EnclaveDeleteResponseTimeRatio) we can learn something about the WLM configuration. But before we can use this data we have to look slightly more to the right at TransactionClass.

If some requests (or all of them) have a TransactionClass name then potentially those requests are going to end up in different WLM service classes. I won't go into all the WLM details here, but
suffice to say that what we're about to do with Response Time Ratio only works per Transaction Class unless you know more about the WLM classification rules being used.

For our data we have no Transaction Class names. That means Don isn't using a classification XML file to provide transaction class names for different URLs. And every request is ending up in the same service class based on whatever classification rule applies to the server cluster name (way back in column D).

So all that aside, what is this Response Time Ratio thing and what does it tell us? WLM calculates the Response Time Ratio by dividing the actual response time (enclave create to enclave delete) by the GOAL for this service class and multiplying by 100. So basically its a percentage of the goal. Note that WLM caps the value at 1000 so if we really blew our goal you might see some big numbers but they won't get bigger than 1000 no matter how badly we did.

So let's look at our data. If we sort the data by this column in descending order we can see that the worst we ever did was a value of 896. It falls off pretty fast and gets under 100 for the vast majority of our requests. So overall we did pretty well compared to our goal.

But what is the goal? Ah...look down a few rows and find a request that has a Response Time Ratio of 100 (you can do this trick with a different value but you have to normalize the results to 100..its much easier if you can find something around 100). A value of 100 means that this request EXACTLY met our response time goal for this service class. So how long did it take to run this request? We'll get to timestamp analysis later, but if we just quickly subtract column AC (Work Received) from column AN (request complete) we get roughly the value WLM used for response time for this request. Sorted in descending order by response time we have a value of 100 in row 12 and the AC and AN values are 3581344774481 and 3581344774532 respectively. Subtracting we get 51. That's in milliseconds. The timestamps for 'complete' isn't exactly where the enclave is deleted so we are probably off a little. And it is unlikely Don's system has 51 milliseconds configured as the goal for this service class. People like round numbers. So its probably 50 milliseconds.

I asked Don to check and he reported "We are using the default CBCLASS, which has "99% complete within 00:00:00.050"."

Timeouts and Other Odds and Ends

What other configuration stuff can we find? Column CJ has the Timeout value. This is the dispatch timeout used for the request. Depending on configuration, especially if you are using the classification XML file, figuring out what the timeout value was can be tricky. SMF knows the value the server used and now you know too. Column CI has the queue timeout. If you have configured the queue-timeout-percentage then this will be the configured percentage of the dispatch timeout value in column CJ. If you haven't configured the percentage, we still knock a little bit off so the queue timeout value is, in this case, 297, when the timeout value is 300. If you look closely you'll find the one internal request and it has a dispatch timeout of 1200. Internal requests are generally handled as IIOP requests for timeout purposes, so the IIOP dispatch timeout is probably set to 1200 (although there are a couple of ways to do that).

We can also see that for a timeout the Thread Hang Recovery code will collect a traceback of the dispatch thread as part of trying to get the request to complete (Column CK). A CPU timeout value is not configured and neither is the Dispatch Progress Monitor (DPM).

We can also see that for the HTTP requests the timeout recovery action is set to '1' which is SERVANT (a value of '2' is SESSION).

We can also see the outbound IIOP request timeout is configured to be 180 seconds.
And finally we don't have a value for the message tag for any of these requests which would be set in the classification XML file.

### Application Information and Users

What application(s) are we running? If you are going to do analysis of application performance you need to be able to separate things out by application in case the server you are looking at is hosting more than one (or the one application has more than one part). Since we have all HTTP requests in our example we can look over at column 'CA' and see the URI sent to us. In this case all our requests are for SuperSnoop. But you can sort by this column and easily see what different URIs were received and how many. As you do later analysis you may want to keep the data sorted by URI so you can compare within applications appropriately.

The other thing you might want to know is WHO is running these requests. Don apparently doesn't have the optional security section turned on, but if he did you might see some identity information showing up in columns CB, CC, and CD.

But we aren't entirely lost. Back up a bit and look at column BK. That's 'Origin'. Here we see the host/port that originated this request. For our data Don is just showing us the ip address of the system that is driving his test load. If you had real users you could sort the data by the IP address and perhaps determine where in your network different types or requests are originating. Or if problem requests all come from the same segment.

For our data the IP address is always the same but the port numbers are different. There are quite a few different ports used. I suspect if we sort by this column and then secondarily sort by request arrival time we can probably see groupings that would tell us how many requests were driven using a particular port before it was closed and a new connection created. The port numbers get reused so some are used more than others, but time-grouping them would probably separate them. I decided to leave that as an exercise for the reader :-)

### Errors Twist The Tale

I probably should have brought this up earlier, but before you start digging into the data and reaching dramatic conclusions you need to make sure nothing bad has happened. A timeout or a servant region recycling due to an abend of some sort will skew the data. Especially if you don't throw out the rows for requests that never finished execution because they are missing critical data that is collected at end-of-dispatch.

The easiest way to find these problems is to sort the data in descending order by column Z (MinorCode). Any non-zero minor code indicates a problem. You can look them up in the information center but the ones I see most often relate to timeouts and dead/dying servant regions. You can look at the timestamps and figure out when this happened and perhaps separate your analysis into before and after, if it's that simple an edge.

### Times and Math

We'll just look briefly at one more thing in the big spreadsheet. Well, really several things: the time values. If you look around column AB (Received Time) you will see a series of timestamps in different formats. We provide five different timestamps: Received, Queued, Dispatch Begins, Dispatch Ends, and Request Complete. For each time we provide the value in a readable form and in milliseconds. Milliseconds since when? Well, since when the MVS TOD timer started. But that
What I Can Tell About You From Your WebSphere SMF 120-9 Records

really doesn't matter. The millisecond values are really only interesting if you want to do math with the timestamps (which is hard to do with those human-readable values).

To help you with that we've also done some math and provided columns that are the difference between each pair of times. For example, in column AF “InCr(ms)” we have calculated the difference between the 'Received' and 'Queued' timestamps.

Some of the analysis on the timestamps we'll save for later with specialized reports, but you can glean a lot just browsing the raw spreadsheet. For example, sort by column AI (InQueue) in descending order to find if any requests spent any significant time on the queue. Sort by Column AL (InDispatch) to find the longest (or shortest) dispatch time. When looking at dispatch times remember too that you might need to also sort by URI to get separation by application.

What does Don's data look like? Sorting by InQueue we see that three requests took 10 or more milliseconds to get into a servant, but most took just 1 millisecond (which is probably rounded up).

If we sort by InDispatch we see two requests that took over 300 milliseconds to execute but most took less than 10. So why the two really long ones? Well, if we sort the data by actual dispatch-begin timestamp we see that the long requests were the first and third to begin dispatch. Furthermore, scrolling to the right to Servant JobID (AQ) we see these were the first requests to run in the two servant regions we have. These are freshly started servers (I know that because Don told me). So there is probably some sort of first-execution processing that takes place that extends the dispatch time for the first request. If we look at TCB CPU time (Column Y) we can also see that these requests used substantially more CPU time than other requests.

If you want to look at other time differences such as complete response time by subtracting Received Time from Request Complete (AN-AC) you can easily create your own columns and do the math. And use built-in spreadsheet functions to average or find min/max values for whole columns etc. Just be sure when you process a whole column that everything you capture is the same (same application and/or no internal requests etc).

I've Got Affinity For You

An application can create session data. For example, an HTTP application can create an HTTPSession object. This object lives in the servant region and can be accessed by subsequent requests from the same client. A shopping cart is a good example of an HTTPSession object.

Having session data means subsequent requests have to be routed back to the same servant region. The session data also takes up space in the JVM Heap. Session data which is unused will be automatically freed if it is untouched for a configurable amount of time.

As you were looking around in the spreadsheet you might have noticed those big nasty hex strings over in columns CF and CH. Those are the obtained and routing affinity tokens. When session data is created an affinity token flows back to the client. That is the obtained token. When a subsequent request comes in that wants to use that session data it sends its affinity token in with the request. That shows up in SMF as a routing token.

I should point out here that the affinity recording data only shows up in records written by WAS V8 and up. If you're still on V7 you won't get this data.

You can use this information from the SMF 120-9 to find out how affinities are being created and used. But its a little tough to work with in the basic spreadsheet. So we created a couple of other plugins to help out.

The first is the AffinityReport. This report finds all the obtained tokens and tracks down all the requests that use it as a routing token. It shows the time the token was created and the URI of the
request that did it. It also shows the servant region where the affinity was created so you can tell if they are balancing.

In the data from Don we can see 10 affinities got created and each one was used 499 times. The affinities were all created at about the same time, no doubt the beginning of the run.

We can also see that the affinities divided evenly (five each) between the two servant regions. That means that the stateful-session-placement environment variable is probably set to tell WLM to balance affinities between the servant regions.

For some more detail let's move on to the AffinityCreation report. I've produced this report in three flavors just to show what it can do. Its not very interesting with our data since all the affinities were created in a very short time span.

Let's look first at AffinityCreationHours. This shows, hour by hour, how many affinities were created. Since all our affinities were created within a few seconds, this isn't very interesting. But it does show the spread between the two servant regions very nicely. In a real system with affinities being created all the time and data spanning a whole day you could get a real sense for the use of the application and routing of work.

The AffinityCreationMinutes report is pretty much the same for our data, but again provides a closer look within a particular time frame.

Finally, AffinityCreationSeconds shows our affinities being created across our very short time span. This is only really interesting if you have a tight burst of affinity creation and want to really zoom in.

Session data can tie up storage in the JVM Heap in the servant regions and, if unused, can use up CPU for the cleanup thread removing them when they expire. Used session data affects how requests are routed among servant regions and how workload is balanced. These two reports can give you a good look at what is going on with affinities to help you understand the impact to your server caused by how the application (and the users) are using session data.

---

**Servant Region and Thread Usage**

“How many dispatch threads should I have in my servant region?”

I get that question quite a bit and my typical IBM response is always, “It depends, how many threads are you actually using now?”

The ThreadRequestDensity report can help you answer my question which will help to answer your own. Let's take a look.

This report shows, for each servant region, minute by minute, the number of requests that started dispatch on a particular TCB (thread). We see first of all that we are using two servant regions. We know from our affinity analysis that this is probably because WLM has been told to spread the affinities out. Clearly it isn't because one servant region's threads can't handle the work.

As it happens, these servant regions are configured with 12 dispatch threads each. You can't tell that from the SMF data. I had to ask Don.

In one servant region we only used two threads and in the other only three. And, in fact, if you look closely you will see that sometimes in the first servant we really only used one thread. And in the second servant region we used two threads most of the time but part way through the run switched to a third thread leaving one of the original ones idle. I think it has to do with the timing of things and tiny little spikes in the incoming work.

Of course one explanation could be that the last thing that thread started to run didn't finish. The report shows work beginning dispatch so if the last thing took a long time end then it would explain why that thread didn't pick up anything new. Of course we know from looking at dispatch times in
the big spreadsheet that nothing ran for minutes, but another approach is to run the ThreadRequestDensity2 report. This does the same thing, but reports on dispatches ending in that minute instead of dispatches starting.

Toggling back and forth between the two reports we see there are no differences. So we don't have any requests beginning in one minute and ending in a later minute.

What else can we learn from these reports? Looking at the totals minute by minute we can see again that work was pretty balanced between the servants and averaged around 140 requests per minute. Looking at the thread totals we can see that in each servant region one thread clearly handled the majority of the work with some of it spilling to a second thread as needed. This is a direct consequence of how fast the work was coming in from how many clients.

Which leads us to our next report.

What Was Where and When?

The RequestDensity report goes second by second through the interval for which we have data. For each second it looks at each request and determines if the request had arrived in the controller, had arrived in the queue, had arrived in dispatch, or had arrived in completion during that second.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>3</td>
<td>6:59:04</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>5</td>
<td>6:59:05</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>7</td>
<td>6:59:06</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>9</td>
<td>6:59:07</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>11</td>
<td>6:59:08</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
<td>GMT</td>
</tr>
<tr>
<td>13</td>
<td>6:59:09</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Each row represents a one second slice of time

Columns shows the number of requests that entered each state for the second of time represented by the row in the spreadsheet

From the WP102311_RequestDensity.csv file in the accompanying ZIP file

Since these requests all run in well under a second, it is quite common for a request to be seen arriving in all four states within a single one-second slice of time:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Thursday</td>
<td>2013</td>
<td>5:59</td>
<td>PM</td>
</tr>
<tr>
<td>3</td>
<td>6:59:04</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

What report would look like for hypothetical single request that processed through all four states in the one second time slice shown for each row in the report

Note that this report is showing you how many requests ENTERED the state in that second, not how many are IN that state.
Looking at the report we can see at 5:59:04 (the first row) we had 3 requests arriving in the CR, 3 arriving in the queue, 3 arriving in dispatch, and 2 arriving in completion:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td>In CR</td>
<td>In Q</td>
<td>In Disp</td>
<td>In Comp</td>
</tr>
<tr>
<td>2</td>
<td>Thursday June 27 2013 5:59:04 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

From the WP102311_RequestDensity.csv file in the accompanying ZIP file

This does *not* mean we had 11 requests in the server. In this case it means two requests made it all the way through the server within this one-second slice of time. A third request didn’t quite finish, as indicated by the value 2 in the completion state column.

The next second of time in the report (5:59:05) shows similar numbers (threes across) except with a four 'in completion'. That's the third request from the first one-second time slice finishing up.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td>In CR</td>
<td>In Q</td>
<td>In Disp</td>
<td>In Comp</td>
</tr>
<tr>
<td>2</td>
<td>Thursday June 27 2013 5:59:05 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Thursday June 27 2013 5:59:05 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

From the WP102311_RequestDensity.csv file in the accompanying ZIP file

So what does this tell us? I mostly look at what is essentially the arrival *rate*, which is the first column of numbers. Averaging these we find we had about 5 requests turning up per second over all the one second time slices in the report. A huge spike or sudden dip in arriving requests might tell us something about whatever is driving (or routing) requests to this server.

The next interesting thing to look at is the 'in queue' numbers. This is how many requests were seen being put on the WLM queue during this one-second period of time. It is *not* necessarily the queue depth because requests can be put on the queue and taken off the queue within the same one-second period of time.

Looking at the differences between the In CR and In Queue and In Dispatch numbers we find they track almost exactly. That means the server is keeping up very nicely.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td>In CR</td>
<td>In Q</td>
<td>In Disp</td>
<td>In Comp</td>
</tr>
<tr>
<td>2</td>
<td>Thursday June 27 2013 5:59:04 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Thursday June 27 2013 5:59:05 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Thursday June 27 2013 5:59:07 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Thursday June 27 2013 5:59:08 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Thursday June 27 2013 5:59:09 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>Thursday June 27 2013 5:59:11 PM GMT</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hypothetical report showing what a spike in arrival rate would look like

The next interesting thing to look at is the 'in queue' numbers. This is how many requests were seen being put on the WLM queue during this one-second period of time. It is *not* necessarily the queue depth because requests can be put on the queue and taken off the queue within the same one-second period of time.

Looking at the differences between the In CR and In Queue and In Dispatch numbers we find they track almost exactly. That means the server is keeping up very nicely.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time</td>
<td>In CR</td>
<td>In Q</td>
<td>In Disp</td>
<td>In Comp</td>
</tr>
<tr>
<td>2</td>
<td>Thursday June 27 2013 5:59:04 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Thursday June 27 2013 5:59:05 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Thursday June 27 2013 5:59:06 PM GMT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Thursday June 27 2013 5:59:07 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Thursday June 27 2013 5:59:08 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Thursday June 27 2013 5:59:09 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>Thursday June 27 2013 5:59:10 PM GMT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>Thursday June 27 2013 5:59:12 PM GMT</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

From the WP102311_RequestDensity.csv file in the accompanying ZIP file

© 2013, IBM Software Group, Application and Integration
Middleware Software, Poughkeepsie, NY
- 10 -
Version Date: Tuesday, October 15, 2013
Another report is called RequestDensity2. An example of this is not provided in the accompanying ZIP file. This report increments the counters a little differently. It increments the counters for the seconds that the request is in a particular state (as opposed to RequestDensity, which increments counters when the requests ENTER a particular state).

This can show you things like the WLM queue depth since the per-second counters are updated for every second a request is in the queue. *But this approach has its own challenges.* For example, imagine a request that remains on the queue for several seconds and then gets dispatched ¼ of a second into the following one-second time slice. Do you count that as being "in" the queue? Or "in" dispatch? Or perhaps double-count the request as being in "queue" and "dispatch"?

For RequestDensity2 it's possible the report may not show the request as 'on the queue' for the one-second time slice where it was on the queue and then dispatched. I felt that trying to account for this might result in some over-counting that would be (I thought) more confusing than the potential undercounting we have now.

### Show Me The Money....CPU Time

There is a lot of information about CPU usage in the SMF 120-9 record. In this section we will look at how to work with the different values and do some closer analysis.

#### Enclave CPU vs. TCB CPU

The SMF 120-9 record includes TCB CPU time, enclave delete CPU time, and enclave CPU time 'so far'.

The 'so far' values are acquired by calling the WLM IWMEQTME API. It turns out these values are current as of the last time data was posted to the enclave. This means they aren't necessarily current as of the time the API was called. Sometimes they make sense and sometimes they don't. I generally ignore them completely.

The enclave-delete values are returned to WebSphere by WLM when we delete the enclave. They break out the CPU used in regular GP, zIIP and zAAP. You can get zIIP CPU time from a WAS request when the customer has configured zAAP-on-zIIP. The values returned are a little funky. To get them into milliseconds you need to divide by 4096.

By contrast, the TCB CPU times are already in milliseconds and are reported as total CPU time (all processor types) and speciality engine (zAAP/zIIP) CPU time. You have to subtract to get general processor CPU time.

Finally, realize that the TCB CPU times and enclave CPU times are captured at slightly different spots in the code, so the values won't match exactly. But they should be pretty close unless the enclave was used for things that didn't run on this TCB (async beans, propagated via an IIOP request to another local server).

#### Dispatch Thread Usage

The last special report we will look at is the DispatchThreadUsage report. This report totals up all the time (elapsed and CPU) that was spent dispatching requests on each TCB. It then calculates the percentage of the total measured time that each thread was dispatching a request and the percentage of that time that was spent using CPU.

Let's look at our sample data. Here again we see our five TCBs that dispatched all the work. The 'TimeRange' column is the window from the start of the first request to the end of the last request for which we have data. The range is different for each servant region.
As before, we can see that in STC04664 two threads did all the work and one spent more time than the other, since it did more work. Meanwhile in STC04665 one thread did most of the work while the rest was split between two other threads.

In all cases the load was pretty light so the time spent in dispatch was a less-than-one percent of the total elapsed time (Column G).

The total CPU time per thread is interesting, but the next column over is more interesting. This shows the percentage of the elapsed dispatch time that was also CPU time. In other words, when the thread had something to do, how much of that time was spent doing something?

In this case, most of the time, the numbers for all threads are around 70%. I've seen this report from customer data where this column indicated CPU was in use only one or two percent of the time. That would indicate that most of the time the dispatched request was waiting for something. You can't tell what from this data though. It could be waiting for the CPU. Or it could be held up by lock contention. Or it could have sent a request over TCP/IP to some other server or database and is waiting for a response. Or probably other things. But you can at least tell from this if it was or wasn't waiting.

Conclusions

I'll just close by saying we've only just scratched the surface of the analysis that can be done using the data available in the SMF 120-9 records. If you write your own plugins or have some other insight to add please feel free to suggest updates to this or the 'plugins' techdoc. I can't guarantee suggested plugins will get implemented of course….unless you send them to me already-written :-) Thanks for reading this and good luck with your own SMF data.
## Document change history

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 27, 2013</td>
<td>Initial Version</td>
</tr>
<tr>
<td>July 24, 2013</td>
<td>Update with document number</td>
</tr>
<tr>
<td>July 30, 2013</td>
<td>Clarified some points about V8-only content</td>
</tr>
<tr>
<td>September 27, 2013</td>
<td>Added some text about RequestDensity2</td>
</tr>
<tr>
<td>October 14, 2013</td>
<td>Corrections and clarifications per comments from Carl Farkas, the &quot;What was where when&quot; section re-engineered to be vastly easier to understand – work done by Don Bagwell.</td>
</tr>
</tbody>
</table>

---

End of WP102311