Using DMS with Oracle 9iAS1.0.2.2

Carol Orange and Bruce Irvin

1.0 Introduction

The Dynamic Monitoring Service (DMS) within Oracle 9iAS automatically measures runtime performance statistics for Oracle HTTP Server processes and JServ processes. The iAS performance metrics are measured automatically and continuously using efficient performance instrumentation hooks inserted into the core implementations of the HTTP and JServ servers. No extra configuration is required to enable the measurement of the statistics.

The data collected allows you to monitor the duration of important phases of request processing as well as status information such as the number of requests being handled at any given moment. This information can be used while troubleshooting to help locate bottlenecks, to identify resource availability issues, or to help tune your webserver to achieve the maximum throughput and the minimum response time possible.

This document describes how to use the 9iAS performance metrics. In particular, Section 2 explains how to extract and interpret 9iAS server statistics while Section 3 describes how these statistics can be used for monitoring and troubleshooting. Appendix A describes the organization of the performance metrics and provides a complete listing of the statistics available in 9iAS. Appendix B describes extra configuration steps needed for some iAS sites. Appendix C documents the tools provided for extracting the performance metrics from 9iAS servers.

2.0 Monitoring iAS with DMS

There is no GUI available for extracting the DMS metrics available in iAS 1.0.2.2. An Oracle Enterprise Manager display for 9iAS performance metrics will be available with 9iAS Version 2.0. Version 1.0.2.2 does include some simple tools which are documented in Appendix C.

The tools include:

The Spy Service. This pre-packaged servlet can be accessed from any web browser. It is used in the majority of the examples in this document. (See Appendix C.1)

Flexmon. This is a command line tool which can be used to monitor a specific set of statistics over a period of time. (See Appendix C.2)

dmsGrab. This simple Java program can be used to grab all the statistics on your site. We recommend that you use dmsGrab to collect all the statistics on your site periodically (say every 15 minutes). Full statistics dumps are very usefule for troubleshooting performance problems on live systems, archiving statistics for offline analysis, and establishing performance baselines (See Appendix C.3)

2.1 HTTP Server Statistics

In this section, we describe how you can extract and interpret the DMS statistics gathered for the HTTP daemon processes running on your server. This section makes extensive use of the Spy service documented in Appendix C.1.

Note. It is important to realize that while HTTP Server uses multiple HTTP daemon processes to service requests, the statistics presented to you are summarized for all of the daemon processes spawned by HTTP Server since the web server was started.

Before you begin, restart the HTTP Server (**apachectl restart**)

You can use the following URL to request all of the metrics collected for HTTP Server:

```
http://myhost:myport/dms0/Spy
```

You should see statistics for each of the HTTP Server modules on your system, as well as summary statistics which have been gathered from the HTTP daemon processes running on your host.

To see only the metrics and metric sets directly contained under the "/Apache" name, modify the request as follows:

```
http://myhost:myport/dms0/Spy?recurse=children&name=/Apache
```

The response shows the **/Apache/Modules**, and several statistics. The statistics groups (see Appendices A.1 and A.2) describe the following:

`connection`:

The **`connection`** phase, starting from the time an HTTP connection is established to the time it is closed.

`request`:

The phase during which an HTTP Server daemon reads a request and sends a response for it (first byte in, last byte out). There may be more than one **`request`** serviced during a single **`connection`** phase. This would be the case if the HTTP parameter KeepAlive were set and utilized by the client.

`handle`:

The phase in which a request is handled by an HTTP Server module. Note that a single request may be handled by more than one HTTP Server module. The handle statistics presented under /Apache are summarized for all of the HTTP Server modules.

Note. The **`.avg`** and **`.time`** statistics produced for the phases above increase from handle to request to connection. The time relationship among these three phases of managing a user request in the HTTP Server is depicted in Figure 1 below.

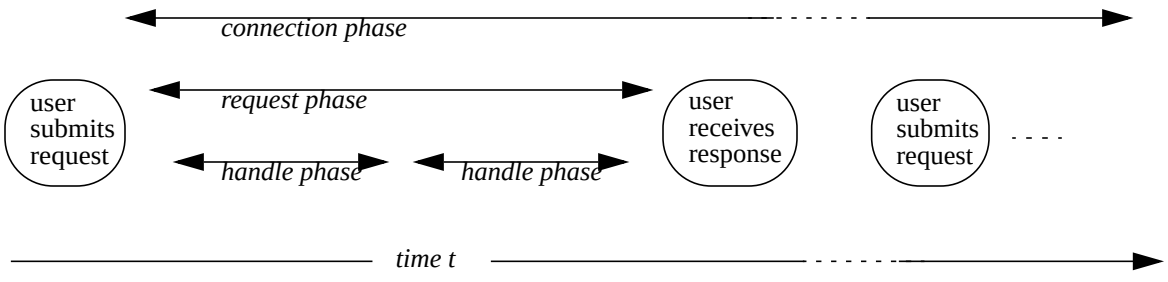

FIGURE 1. The relationship among the phase metrics collected for /Apache.

If KeepAlive is on and clients use it, the duration of a connection may be much longer than the time required to perform a request and return a response. (See Figure 1.) This is because the connection may remain open while a single client submits multiple requests.

The metrics for these phases are fully documented in Appendix A.2. We can view data collected for the HTTP Server modules by modifying the request:

```
http://myhost:myport/dms0/Spy?recurse=children&name=/Apache/Modules
```

The response shows the list of modules supported in the HTTP Server . The numMods.value indicates the number of modules ready configured in the server. Before studying the statistics, generate a small amount of server activity by requesting the following URL a few times:

```
http://myhost:myport/servlets/IsItWorking
```

Accessing a servlet such as "IsItWorking" causes activity in the **mod_jserv** module within the HTTP server. To see the **mod_jserv** metric values, request this URL:

```
http://myhost:myport/dms0/Spy?name=/Apache/Modules/mod_jserv.c
```

The response includes **handle.*** metrics specific to this module. If you inspect the handle statistics for most other modules, they will be 0. This indicates that most other modules have not been used in any requests.

The **mod_dms.c** metrics, however, will be nonzero. Try:

```
http://myhost:myport/dms0/Spy?name=/Apache/Modules/mod_dms.c
```

Note that it shows that one request is active because **mod_dms.c** is the HTTP Server module that formats the dms0/Spy metric dump pages. It is indeed active while it is formatting its own page.

Section 3.4, explains how to use the HTTP Server statistics to when monitoring or troubleshooting a 9iAS site.

The HTTP Server metrics also include some internal metrics. You can see them with a request like this:

```
http://myhost:myport/dms0/Spy?name=/DMS_Internal
```

You'll see a few system statistics that may be useful in troubleshooting. For example, if your CPU usage (**cpuTime.value**) growth is nonlinear with respect to the number of requests being handled (**request.completed**), you may have a poor system configuration, or some module may be using application logic that scales badly. You can monitor different combinations of metrics over time using flexmon as described in Appendix C.2.

2.2 JServ Metrics.

Now let's look at the statistics produced for each JServ process. In order to generate some data, we suggest you first use the following URL a few times (if you haven't already):

```
http://myhost:myport/servlets/IsItWorking
```

To see the full set of JServ statistics, try a URL like this in your web browser:

```
http://myhost:myport/servlet/Spy
```

You will be presented with a set of statistics for each servlet or JSP request handled by the server. You will also be presented with a set of statistics for each zone which contains a servlet that has been called. For example, since the **IsItWorking** and **Spy** servlets are available in the root zone by default, this URL will result in statistics for both the root zone and for these servlets. Statistics for the JServ process will also be presented.

Let's look briefly at the JVM metrics before diving into the jserv metrics. Try the following URL:

```
http://myhost:myport/servlet/Spy?name=/JVM&units=true
```

In troubleshooting problems where resource limitations may be an issue, the **JVM** metrics can clearly be of use.

Note. The metrics described in this section are fully documented in Appendix A.3.

2.2.1 The "jserv" Metrics

Now let's look at the statistics that are located immediately under the /jserv name.

```
http://myhost:myport/servlet/Spy?recurse=children&name=/jserv
```

After you have made one or more requests to the **IsitWorking** and **Spy** servlets,the output will look similar to the following:

```
<DMSDUMP version='2.0' timestamp='984164727768 (Fri Mar 09 11:05:27 PST
2001) id='1' name='JServ'>
<statistics>
 /jserv [type=JServ_Server]
  maxBacklog.value:  5 requests
  maxConnections.value: 50 threads
  host.value:   localhost/127.0.0.1
  port.value:   8007
  readRequest.active:   0
  readRequest.avg:      0.8181818181818182 msecs
  readRequest.maxTime:      9 msecs
  readRequest.minTime:      0 msecs
  readRequest.completed:        33 ops
  readRequest.time:     27 msecs
  activeConnections.maxValue:       1.0 threads
  activeConnections.value:      1 threads
  idlePeriod.maxTime:       7253 msecs
  idlePeriod.minTime:       532 msecs
  idlePeriod.completed: 33 ops
  idlePeriod.time:      98961 msecs
  /jserv/OPM [type=OPM]
  /jserv/root [type=JServ_Zone]
</statistics> </DMSDUMP>
```

The output shows the **root** zone and a list of metrics including the following:

port.value

the TCP/IP port on which this JServ process is listening.

readRequest.*

This set of metrics measures the interval in which JServ reads a request using the ajpv12 protocol in preparation of processing. As can be seen, the average time to read a request is sub-millisecond on our

not very loaded Sun Solaris system. High numbers (relative to normal) for this metric may indicate a network problem.

`maxConnections.value`

The number of concurrent requests that may be handled by this process.

`activeConnections.value`

The number of requests being handled at the time of this request.

`activeConnections.max`

The maximum number of requests that have been handled concurrently since this process was started. If this number is greater than 10, and you are not CPU bound, it may be useful to balance across more JServ processes. See Chapters 3 and 5 of [1] for guidance.

`idlePeriod.*`

Information about the duration and number of times the server has not had any requests.

`host.value`

the host this process is running on.

`maxBacklog.value`

maximum number of requests that may be queued in the OS waiting for this JServ.

2.2.2 The Zone Metrics (/jserv/root)

Now let's take a look at the information available for zones. In our example, we have configured the root zone, so we will modify the request we performed to get /jserv metrics to get the metrics for the root zone:

```
http://myhost:myport/servlet/Spy?recurse=children&name=/jserv/root
```

In this case, the output will include the following:

```
/jserv/root/IsItWorking [type=Servlet]
/jserv/root/Spy [type=Servlet]
checkReload.active:  0
checkReload.avg:     0.3333333333333333 msecs
checkReload.maxTime:     1 msecs
checkReload.minTime:     0 msecs
checkReload.completed:       3 ops
checkReload.time:    1 msecs
activeSessions.value:        0 sessions
readSession.count:   0 ops
writeSession.count:  0 ops
loadFailed.count:    0 ops
```

In our example, the **`IsItWorking`** and **`Spy`** servlets have been running. The response's zone metrics can be interpreted as follows:

`checkReload.*`

This measures the time to check whether a called servlet needs to be reloaded before it is run. As the number of concurrent requests increases, so does the average time required to check on reloading (this is due to synchronization in the JServ code). If this number becomes very high in comparison to the time required for servlet processing, it may be useful to balance the load among more JServ processes.

`activeSessions.value`

The number of sessions which have been created, but not invalidated.

`readSession.count`

The number of times session data has been read in this zone.

`writeSession.count`

The number of times session data has been written in this zone. See note below.

`loadFailed.count`

The number of times a servlet with an invalid name was requested in this zone.

Note on session metrics. The session metrics can be valuable for detecting whether sessions are being created, but never used (something that can easily happen for JSP applications). Also, you may notice that an application writes session data that is never read. These metrics should be watched if your system appears to be consuming more memory than expected. See Tip 3 in Section 3.1.2.

2.2.3 The Servlet Metrics

A set of metrics will be created for each servlet that has been initialized in the JServ process. The statistics for the servlet can be requested with the name /jserv/<zone-name>/<servlet-name>. In our case, we have a zone named root.

To generate an example, we set up a single user test in which the IsItWorking demo servlet was continually requested for 10 seconds. We found some interesting information by inspecting the metrics for this servlet as follows:

```
http://myhost:myport/servlet/Spy?name=/jserv/root/IsItWorking
```

In our experiment on a single processor host using one JServ process, this request generated:

```
processRequest.active:      0
processRequest.avg: 3.088524590163934 msecs
processRequest.maxTime: 722 msecs
processRequest.minTime: 0 msecs
processRequest.completed:   2135 ops
processRequest.time:        6594 msecs
serviceRequest.active:      0
serviceRequest.avg: 0.9334894613583138 msecs
serviceRequest.maxTime: 250 msecs
serviceRequest.minTime: 0 msecs
serviceRequest.completed:   2135 ops
serviceRequest.time:        1993 msecs
loadServlet.active: 0
loadServlet.avg:    0.012177985948477752 msecs
loadServlet.maxTime:    21 msecs
loadServlet.minTime:    0 msecs
loadServlet.completed:      2135 ops
loadServlet.time:   26 msecs
loadServletClasses.active:  0
loadServletClasses.avg:     21.0 msecs
loadServletClasses.maxTime:     21 msecs
loadServletClasses.minTime:     21 msecs
loadServletClasses.completed:       1 ops
loadServletClasses.time:    21 msecs
createSession.active:       0
createSession.avg:  0.0 msecs
createSession.maxTime:  0 msecs
createSession.minTime:  0 msecs
```

```
createSession.completed:    0 ops
createSession.time: 0 msecs
```

These metrics can provide much insight as to the distribution of request processing. This is what each set means:

`processRequest.*`

This metric group measures the duration of all the processing required for this servlet. Nearly all processing phases take place within the **`processRequest`** phase. This includes those measured for the zone, with the exception of **`readRequest.*`**, which measures the time to read the request from HTTP Server. See Figure 2 to understand how the JServ processing phases are related.

`serviceRequest.*`

This group of metrics measures the time spent in the servlet's service method. Thus it measures the duration of the application code (which may include requests to create, read, write, or destroy sessions). If the servlet makes use of any external services, such as a database, then the time reported will include the time required for the external services.

`loadServlet.*`

The **`loadServlet`** metric group measures how long it takes JServ to get a servlet instance which can be used to service this request. This may include loading the servlet from disk.

`loadServletClasses.*`

This metric group measures how long it takes to load the servlet class from disk. In many cases, this will only happen once (**`loadServletClasses.completed = 1`**), as the class is cached in the JServ engine. Note that **`loadServlet.max >= loadServletClasses.max`** is always true as the classes must be loaded from disk at least once.

`createSession.*`

The time required to create a session. **`createSession.completed`** shows the number of sessions that have been created for this application.

`maxSTMInstances.value`

The number of instances of this SingleThreadModel servlet which have been loaded for servicing requests. This value may increase as the JServ process runs depending on the SingleThreadModel servlet settings in the properties file for this zone. (See note below.)

`activeSTMInstances.max`

The maximum number of SingleThreadModel servlet instances which have been used concurrently since the server was started.

`activeSTMInstances.value`

The number of SingleThreadModel servlet instances currently in use.

Note. The *STM* metrics do not appear for the IsItWorking and Spy servlets because neither servlet implements the SingleThreadModel interface. They will appear for any servlet that does.

JSP Metrics. The same metrics that are shown for servlets are also available for JSP applications. However the statistics are split up showing the metrics **`loadServlet.*`** and **`loadServletClasses.*`** under the **`oracle.jsp.JspServlet`**, and the remainder of the metrics under the individual JSP name. To see this try the lottery demonstration JSP, and then look at the statistics under **`/jserv/root/oracle.jsp.JspServlet`**.

```
http:/myhost:myport/demo/basic/lottery/lotto.jsp

http://myhost:myport/servlet/Spy?&name=/jserv/root/oracle.jsp.JspServlet
```

2.2.4 Phase Metric Time Relationships

In Section 2.4, we showed the time relationships among the various HTTP Server metrics in Figure 1. In Figure 2 below, we show the relationships among the phase metrics measured in JServ.

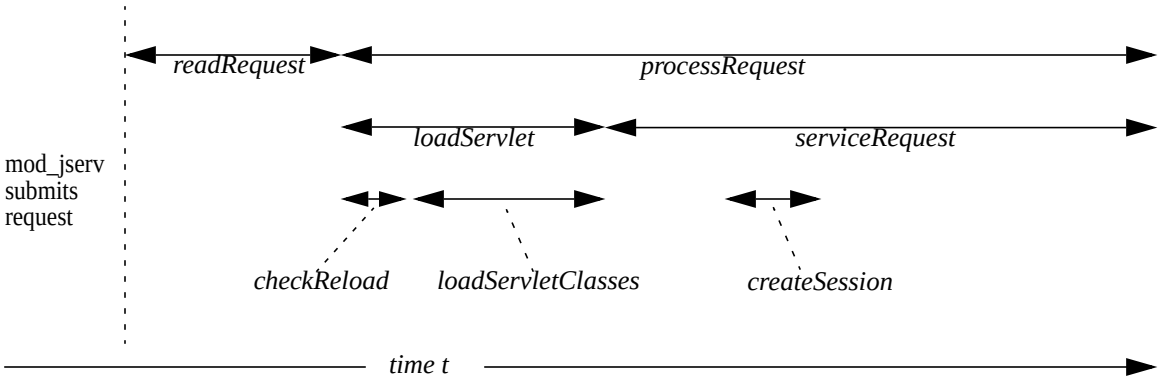

FIGURE 2. JServ Request phases monitored with DMS

3.0 Monitoring and Troubleshooting your site with DMS

This section provides tips for debugging performance problems on your webserver. To make best use of these tips, you should use the dmsGrab utility described in Appendix C.3 to periodically log the iAS metrics. If you periodically collect the metric values, then you can compare “normal case” behavior with values collected during debugging of performance anamolies.

3.1 General tips

There are various sets of metrics which describe resource limits and resource usage. If your system is optimally configured, you should see some difference between the two. If this is not the case, you may be risking a bottleneck during peak usage. You should also see some limit in resource usage growth, as in most cases, resources should be reused or released after use. In this section, we discuss some of the things that might signal resource consumption problems on your system, and what you can do about them.

3.1.1 Resources nearing their limits

In JServ, there are a couple of metric pairs that should be monitored periodically to be certain your system is appropriately tuned.

Tip 1. Monitor `/jserv/maxConnections.value` versus `/jserv/activeConnections.*`

The **`security.maxConnections`** parameter in your **`jserv.properties`** file limits the number of requests the JServ process can handle concurrently. The metric, **`/jserv/maxConnections.value`**, reports the value of that parameter. The metric **`/jserv/activeConnections.value`** reports the number of concurrent requests being handled by the JServ process. This can be monitored over time (see Appendix A.3.2.1 for instructions) to determine whether the configuration is satisfactory to handle the user load.

It is important to know whether the number of concurrent requests nears the maximum possible. This can also be seen by inspecting the metric /**`jserv/activeConnections.max`**, which reports the maximum number of requests that have been handled concurrently since the JServ process was started. If the number of concurrent requests has neared the maximum possible, and the system is not otherwise resource bound (CPU for example, see Section 3.1.2), then you may need to increase the limit on the number of concurrent requests that can be handled on your system. This can be done by increasing **`security.maxConnections`** for this JServ process, or by balancing your load over more processes.

In addition, if you monitor **`/jserv/activeConnections.value`**, then you may have noticed some response time degradation as it increased. If so, it may be useful to lower **`security.maxConnections`** in the properties file for your zone and to distribute the load among additional JServ processes. See Chapters 3 and 5 in [1] for advice on load balancing.

An example of the values produced on a system with the default 9iAS configuration and 20 concurrent users is:

```
maxConnections.value: 50 threads
activeConnections.maxValue: 11.0 threads
```

Even though there are 20 users concurrently making requests, we don't see them all in JServ concurrently, as some of them will be busy on HTTP Server processes and others in transport.

Note. If you see **`maxConnections.value = activeConnections.max`**, you are likely to experience some hangs on your system. This could be due to a resource bottleneck, or you may need to configure your system for more users as indicated above.

Tip 2. Monitor `/jserv/<zone>/<servlet>/maxSTMInstances.value` versus `/jserv/<zone>/<servlet>/activeSTMInstances.max`

The **`singleThreadModelServlet.maximumCapacity`** parameter in the properties file for your zone sets a limit on the number of instances that will be created for a servlet which implements the SingleThreadModel interface. The **`singleThreadModelServlet.minimumCapacity`** parameter determines the minimum number of instances that will be available to service clients. The **`/jserv/<zone>/<servlet>/maxSTMInstances.value`** DMS metric reports the current number of instances that are available to handle requests. Initially, this will be the value of **`singleThreadModelServlet.minimumCapacity`** and can grow as large as **`singleThreadModelServlet.maximumCapacity`**. In configuring your system, the maximum should be set to the maximum number of concurrent requests your JServ process can handle for any of your SingleThreadModel servlets (see Chapter 5 in [1] for guidance in setting the singleThreadModelServlet parameters).

The metric /jserv/<zone>/<servlet>/activeSTMInstances.value describes the number of STM instances being used. This can be monitored over time (use flexmon as described in Appendix C.2.1) to determine whether **`singleThreadModelServlet.maximumCapacity`** needs to be increased. Alternatively, if

you monitor **/jserv/<zone>/<servlet>/activeSTMInstances.max**, you can see whether the number of instances used concurrently has been too high (too close to the maximumCapacity) at any point while the server has been running. If this value is very close to the limit, and the system is not otherwise resource bound, then the number of instances for this servlet should be increased to avoid errors under increased loads. This can be accomplished either by increasing the **singleThreadModelServlet.maximumCapacity** in the properties file for this zone, or by increasing the number JServ processes handling servlet requests for this server. Again, see Chapters 3 and 5 in [1] for load balancing guidance.

Example. If we run a single user against a simple servlet which implements the SingleThreadModel interface on the default HTTP Server and JServ configuration, we find:

```
maxSTMInstances.value:      5 instances
activeSTMInstances.maxValue:     2.0 instances
```

And if we run 20 concurrent users against the same servlet on the same configuration, we find

```
maxSTMInstances.value:      10 instances
activeSTMInstances.maxValue:     7.0 instances
```

First note that when we ran with more users the number of available instances was increased from 5 to 10. (In the default configuration, the minimum is 5 and the maximum is 10.) Also notice that when we were only running with one user, that at some point 2 instances of our servlet were in use. This is because the servlet engine sends the response back to the client (who in our test immediately sends another request) before releasing the instance. On our switched 100 Mbit network, the next request from the client can arrive before the servlet instance used to service the last one has been released.

Since the server is CPU bound during the 20 user test, the DMS metrics show that the default setting

```
singleThreadModelServlet.maximumCapacity = 10
```

is sufficient for this server, since at most 7 of the 10 are ever used.

3.1.2 Runaway resources

Resources that are created and not used, or not efficiently reused will almost always generate performance problems on an application server. Lookups may become less efficient, garbage collection may become required much more often than necessary. Inefficient resource utilization can lead to hitting CPU and memory limitations at a much lower user load than necessary. In some cases, DMS can be used to detect inefficient resource usage.

Tip 3. Watch /jserv/<zone>/<servlet>|<jsp>/createSession.* and /jserv/<zone>/*Session.* metrics.

It is unfortunately easy to accidentally generate a JSP which creates a session that is never used. Each time a new user requests the JSP, a new session will be generated, thereby consuming system resources without serving any purpose. It is also easy to inadvertantly write servlets that do not invalidate their sessions. Without source for the application software, you may not know this could be a problem on your host, but sooner or later you would notice a higher consumption of memory than expected. You can see if there are sessions which are not utilized or sessions which are not being properly invalidated after being used with the DMS session metrics.

You can check whether your applications create sessions using **/jserv/<zone>/<servlet>|<jsp>/createSession.*** metrics. If the metric

```
processRequest.completed > 0
```

for this servlet or JSP, and the metric

```
createSession.completed = 0
```

then you need not worry about runaway sessions, as this application does not create any.

However, if the createSession.completed metric is nonzero, then it is useful to check whether the sessions are actually used. To understand how the DMS metrics can be employed to detect whether the sessions are used, consider the following examples.

Example 1. Suppose we have a servlet which uses sessions effectively and invalidates them appropriately. Then we might see a set of metrics such as the following:

Under **`/jserv/<zone>/<servlet>`**:

```
createSession.completed:    3479 ops
```

Under **`/jserv/<zone>`**:

```
activeSessions.value:  2 sessions
readSession.count:   3583 ops
writeSession.count:  3497 ops
```

The fact that 2 sessions are active when more than 3000 have been created indicates that sessions are being invalidated after use. Meanwhile, since the number of reads and writes of session data correlates more or less linearly with the number of sessions created, it appears the application is making good use of the sessions.

Example 2. An application that creates sessions, but never uses it would generate statistics like the following after a series of calls:

Under **`/jserv/<zone>/<servlet>`**:

```
createSession.completed:    500 ops
```

Under **`/jserv/<zone>`**:

```
activeSessions.value:  500 sessions
readSession.count:   0 ops
writeSession.count:  500 ops
```

Clearly such an application makes unnecessary use of resources and it is just a matter of time before it causes memory or CPU consumption problems. Had the application been a JSP in which the page directive:

```
<%@ page ... session=”false” %>
```

had been forgotten, but the sessions were never used or invalidated, you would also have seen:

```
readSession.count:   500 ops
```

In this case, the key thing to watch for is that **`activeSessions.value << createSession.completed`** before the session time out (**`session.timeout`** in the properties file for your zone). If it is, the sessions are probably being used and cleaned up afterwards.

Note. When you think of applications that create but do not invalidate user sessions, consider the following. Suppose your application services 10,000 users at a time (which may translate to under 10 concurrent active requests) and that users are on your system for an average of 10 minutes. This means you have a turnover of about 1000 users per minute. Using the default configuration, a session will be invalidated after it has been inactive for more than 30 minutes (**`session.timeout`** in the properties file for your zone). If your application does not invalidate its sessions, then there will be an average of about 30,000 idle sessions on your system in addition to the 10,000 active sessions. Clearly it is worth the effort to monitor the session related DMS metrics to determine whether this problem occurs on your system.

Tip 4. Watch `/JVM/totalMemory.*` versus `/JVM/freeMemory.*`

Suppose you have a servlet or JSP which slowly uses memory by creating more objects than it needs. Sometimes objects are inadvertently stored in a Hashtable, but never used, for example. Such a servlet can cause an JServ process to run out of memory and shut down. On the other hand, if objects are created that can be reused, excessive garbage collection might become required, that could have been avoided. This will have the effect of producing uneven performance.

Memory leaks such as this can be very hard to detect but with the DMS metrics under **`/JVM`**, you can examine the memory use patterns in your JServ process.

Example. If you are creating many objects, and you monitor the /JVM/freeMemory.value metric over time (use flexmon as documented in Appendix C.2.1), you might see a series of data such as the following:

```
freeMemory.value:       6249 kbytes
freeMemory.value:       4680 kbytes
freeMemory.value:       3551 kbytes
freeMemory.value:       52 kbytes
freeMemory.value:       1606 kbytes
freeMemory.value:       2269 kbytes
freeMemory.value:       1962 kbytes
freeMemory.value:       2897 kbytes
freeMemory.value:       572 kbytes
freeMemory.value:       2389 kbytes
freeMemory.value:       4191 kbytes
freeMemory.value:       3652 kbytes
freeMemory.value:       3552 kbytes
freeMemory.value:       6810 kbytes
freeMemory.value:       6562 kbytes
freeMemory.value:       6345 kbytes
```

Note that the amount of free memory is very erratic. This is because the system is being forced to garbage collect due to the fast creation and short lives of objects. You can generally narrow down the problematic application by examining usage patterns during a time series. In particular, you want to find those applications for which the **`/jserv/<zone>/servlet/processRequest.completed`** is increasing. Of course, the application may very well be a JSP. In fact, one common cause for this behavior is the inadvertent creation of sessions in JSPs. See Tip 3 above.

If the system runs out of memory, the JServ process will shut down. This will happen if references to the objects are not released. For example if objects are stored in a Hashtable or Vector and never again removed.

It is of course possible that your process actually needs to use a lot of memory whether or not it stores the data for a time. In this case, the DMS metrics shown above would indicate that the maximum heap size for this process should be increased to avoid frequent garbage collection. See [1] for guidance.

3.1.3 Augment the data

The iAS metrics are even more useful when augmented with other statistics available on your system. Some examples of information that can be of use in performance monitoring and debugging are:

The HTTP Server module mod_status.c. This shows the number and status of HTTP Server daemons on your host. (See Chapter 2 in [1] for help on setting up and using mod_status.)

System monitoring tools. Tools such as **`sar`** and **`vmstat`** on UNIX platforms and the **`Performance Monitor`** on NT should be used to monitor CPU, memory and I/O usage. In various examples, we've mentioned that our interpretation of the metrics assumes that the system is not CPU bound. It is always important to know what the CPU utilization is in understanding performance issues.

Network monitoring tools. Any tool (such as **`netstat`** on UNIX platforms) which can be used to inspect network connections might be of use if you suspect performance problems may be caused by network or TCP configuration issues (See Section 3.2 for some examples).

3.2 JServ Tips

In this section, we discuss some metrics in JServ which should be monitored as a set in order to detect certain kinds of problems.

3.2.1 Monitor Server Request Load

In debugging servlet and JSP problems, it is often useful to know how many requests your JServ processes are servicing. If the problems are performance related, it is always helpful to know if they are aggravated by a high request load. You can track the JServ request load by monitoring:

```
/jserv/activeConnections.*
```

See Section 3.3.1 below for HTTP Server request load monitoring.

3.2.2 Monitor `processRequest.avg` and `serviceRequest.avg`

The metric **`serviceRequest.avg`** reports the average time spent in the application code for a given servlet. If the application is a JSP, then the time reported is that required by the JSP engine **`oracle.jsp.JspServlet`** to service the requested JSP. Meanwhile, the metric **`processRequest.avg`** reports the entire time required to process the request on average, excluding that required to read the request from HTTP Server. This means that

```
processRequest.avg - serviceRequest.avg
```

provides a good estimate of the overhead of the servlet engine. In particular, it provides an excellent indicator of whether the system is optimally configured.

3.2.3 High Load Time

You may find that a servlet application is especially slow the first time it is used after the server is started, or that it is intermittently slow. It is possible that when this happens that the server is heavily loaded, and the response time is suffering as a result. If there is no indication of a high load, however (which you can detect by monitoring your access logs, periodically monitoring CPU utilization, or by tracking the number of users that have active requests in HTTP Server and Jserv using the DMS metrics), then you may just have a servlet that takes a long time to load.

You can see if you have a slow loading servlet if

```
servletRequest.max >> servletRequest.avg
```

and

```
loadServletClasses.avg >> servletRequest.avg.
```

If the time to load the servlet is much higher than the time to service, the first user that accesses the servlet after your system is started will feel the impact. You can avoid this by configuring the system to initialize your servlet when it is started. To arrange this, use the **`servlets.startup`** directive in the properties file for your zone. See [1] for guidance.

Another reason that a servlet might be sporadically slower is that additional instances of SingleThreadModel servlets may need to be created as your user load increases. If you notice that the maximum number of instances used (**`activeSTMInstances.max`**) is greater than the minimum set in the zone properties file

(**singleThreadModelServlet.minimumCapacity**), then you may want to increase the minimum to avoid the overhead of generating additional instances when the server is under load.

3.2.4 JSP Tips

Using the set of DMS metrics provided in iAS , one can only track JSP performance from the perspective of the servlet engine. The key thing that should be watched if you have JSP applications on your system are the session metrics discussed in Tip 3 of Section 3.1.2.

3.3 HTTP Server Tips

Depending on your applications, there are a number of things that you might watch for in your HTTP Server metrics. In this section, we cover the most common uses for this data.

3.3.1 Which Modules are Used?

In troubleshooting, it is often useful to know is which modules are being used on your system. Some requests will be handled by multiple modules, while others by only one. You can see which modules are being used on your system, by inspecting the HTTP Server statistics, and investigating which modules have a nonzero value for any of the **handle.*** statistics. If you have requests which are handled by more than one module, you can detect this on a non-production system by experimenting with the requests individually and inspecting the HTTP Server DMS metrics as you go.

Example 1. If you restart your system and request the home page a number of times using

```
http://myhost:myport/index.html
```

Be sure to fully reload the page into your browser each time. When you inspect the HTTP Server metrics using any of the methods described in Appendix C, you'll find nonzero values for **mod_mmap_static.c** and **mod_include.c**:

```
/Apache/Modules/mod_mmap_static.c
  handle.maxTime: 24
  handle.minTime: 3
  handle.avg: 12
  handle.active: 0
  handle.time: 171
  handle.completed: 14
/Apache/Modules/mod_include.c
  handle.maxTime: 94
  handle.minTime: 3
  handle.avg: 47
  handle.active: 0
  handle.time: 333
  handle.completed: 7
```

If you adjust your request to ask for the home page without the index.html:

```
http://myhost:myport/
```

You'll find nonzero values for **mod_dir.c** and **mod_perl.c** as well:

```
/Apache/Modules/mod_dir.c
  handle.maxTime: 995
  handle.minTime: 777
  handle.avg: 880
  handle.active: 0
  handle.time: 3523
```

```
  handle.completed: 4
/Apache/Modules/mod_perl.c
  handle.maxTime: 1101
  handle.minTime: 967
  handle.avg: 1035
  handle.active: 0
  handle.time: 4142
  handle.completed: 4
```

This is because the HTTP Server must translate a request ending in "/" to the appropriate file name in addition to serving the file. In inspecting the statistics, we see that it is much more costly to determine the correct file name than to serve the file once we have it.

Example 2. We ran a PLSQL "Hello World" application for about 30 seconds. This time, we find the work was all done in **`mod_plsq.c`**:

```
/Apache/Modules/mod_plsql.c
  handle.maxTime: 859330
  handle.minTime: 17099
  handle.avg: 19531
  handle.active: 0
  handle.time: 24023499
  handle.completed:  1230
```

Note that **`handle.max`** is much higher than **`handle.avg`** for this module. This is probably because it is upon the first request that a database connection must be opened. Later requests can make use of the established connection.

3.3.2 Static versus Dynamic Requests

It is important to understand where your server is spending resources, so you can focus your tuning efforts in the areas where the most stands to be gained. In configuring your system, it can be useful to know what percentage of your requests are static and what percentage are dynamic. As seen above, we can see the number of static page requests by monitoring the mod_mmap_static.c module. We determine the dynamic page usage by inspecting the metrics for the type of request we are placing.

Example. Suppose we try the lottery demo supplied with iAS :

```
http:/myhost:myport/demo/basic/lottery/lotto.jsp
```

Use the reload button on your browser to request this a number of times. Then produce a dump of the HTTP Server statistics. You can see the ratio of static and dynamic requests generated by the lotto.jsp by inspecting the statistics for the **`mod_mmap_static.c`** and the **`mod_jserv.c`** HTTP Server modules:

```
/Apache/Modules/mod_mmap_static.c
  handle.maxTime: 78
  handle.minTime: 3
  handle.avg: 9
  handle.active: 0
  handle.time: 503
  handle.completed:  55
/Apache/Modules/mod_jserv.c
  handle.maxTime: 443753
  handle.minTime: 855
  handle.avg: 58245
  handle.active: 0
  handle.time: 1048427
  handle.completed: 18
```

You may notice that for each request to the lottery JSP requires multiple requests to JServ as well as a number of static page requests (the numbers images). Had the lottery demo been an ordinary servlet, we would have only seen one added to **`handle.completed`** in **`mod_jserv.c`** for each request.

3.3.3 HTTP Server and JServ Time Differences

In some cases, you may notice a high discrepancy between the average time to process a request in JServ and the average response time (ART) experienced by the user. If the time is not being spent actually doing the work, then it is probably being spent in transport. As will be clear in the example in the next section, this can point to a couple of different configuration problems. If you notice a large discrepancy, then we advise you check:

1. TCP/IP parameters

2. JServ load balancing configuration

As always, refer to [1] for guidance in these areas.

3.4 Troubleshooting - A Servlet Example

In this section, we use a simple example servlet “BinomialPool” to illustrate some performance problems you might observe on your system, and how you might use DMS to understand them.

The BinomialPool servlet emulates an application which requests an external resource and then does some computation. The external resource is accessed using objects in a pool, and the servlet has to wait for it to do some work before releasing the object. We ran our test on a 100 Mbit switched LAN with 24 concurrent users. Using the default configuration of HTTP Server and JServ delivered with iAS, we achieved:

Average Response Time = 92 msec

We might have been satisfied with this result, but when we ran only one user against this system, we were able to achieve:

Average Response Time = 50 msec

We were therefore concerned with why the ART had deteriorated so much. During the 24 user test, the CPU utilization was about 55%, so we were certainly not resource bound. We also noticed that our throughput increased by only a factor of 12. We employed the DMS statistics (using dmsGrab to get them) to understand where the system spent the time. Here is what we found.

When we looked at the DMS statistics (24 user run) for this servlet on JServ, we saw:

```
/jserv/root/BinomialPool
  processRequest.avg: 49.04076570847113 msecs
  serviceRequest.avg: 47.41904095649962 msecs
```

These statistics are very close to those we collected for the single user run:

```
/jserv/root/BinomialPool
  processRequest.avg: 47.107859531772576 msecs
  serviceRequest.avg: 46.91053511705686 msecs
```

From this we conclude that the JServ process is performing the work of servicing a request very well under load. However, we don’t know where the additional 43 msecs experienced by the client are being spent.

Inspecting the HTTP Server statistics for mod_jserv.c (24 user run) shows:

```
/Apache/Modules/mod_jserv.c
```

```
handle.maxTime: 3459621
handle.minTime: 43703
handle.avg: 84854
handle.active: 0
handle.time: 1334257491
handle.completed: 15724
```

So it seems that the majority of our time is spent in the HTTP Server **`mod_jserv.c`** module, or in communicating with the JServ process. Since it is really the key task of **`mod_jserv.c`** to communicate with the JServ process, we might consider that our TCP/IP parameters need adjusting, or that our JServ load balancing configuration is not optimal.

In considering this, we realize that we had already configured the TCP/IP parameters on our host based on the guidance provided in [1], but we had yet to apply the JServ load balancing guidelines described there. So, we modified our system to use 4 JServ processes with **`security.maxConnections`** set to 10, rather than 1 JServ process with **`security.maxConnections`** set to 50. After making this change, we noticed we were able to make full use of the CPU (100%), and to improve the response time for the 24 users to:

Average Response Time = 80 msec

After making these adjustments, the HTTP Server statistics show:

```
/Apache/Modules/mod_jserv.c
   handle.maxTime: 389994
   handle.minTime: 43670
   handle.avg: 71477
   handle.active: 0
   handle.time: 1286026694
   handle.completed: 17992
```

After setting up load balancing, we see that on average we spend about 13 msec less waiting for a response from an JServ process. We were unable to achieve more improvement in response time because our system was CPU bound. The request had to wait somewhere, and in this case, it was waiting in **`mod_jserv.c`** for an JServ process ready to service it.

Note. On our load balanced system, we were able to service about 2500 more requests in the same time period. Our client side statistics show a 20% gain in throughput.

Appendix A The DMS Metrics in iAS.

The DMS metrics implemented in iAS fall into two distinct structures, one for HTTP Server and the other for JServ. The purpose of the appendix is to provide an easy reference to the structure and meaning of the metrics. Before describing the metric details, we briefly describe the general structure of DMS metrics.

A.1 DMS Metric Groups.

The metrics measured with DMS fall into one of three groups. The values are presented as groups by the various tools, and we discuss them as a group in various parts of this document. In this section, we describe the three types of metrics in DMS:

`Event`

An **`Event`** is always a count of something that happened, and there are no other metrics related to it. Examples of Event metrics in iAS are **`readSession.count`** and **`writeSession.count`** which count the number of times any session has been read or written in a zone.

`State`

A **`State`** metric keeps track of some value. Related to the **`State`** metric **`value`**, are the maximum and minimum values it takes on. An example of a set of State metrics in iAS is the set of metrics **`activeConnections.value`** and **`activeConnections.maxValue`**. The latter tracks the maximum value that is ever reached by the former. In a **`State`** metric, the value is always tracked where the maximum and minimum may or may not be derived.

`PhaseEvent`

In general, a **`PhaseEvent`** measures the time required for some processing. The cumulative **`time`** used in the processing over all the times it has occurred is always measured. There is a set of five additional derived metrics which can be used to understand the performance of some processing phase. Again, not all of these will always be tracked. An example of an HTTP Server set of PhaseEvents is the group of **`request.maxTime, request.minTime, request.avg, request.active, request.time,`** and **`request.completed`**. You can review Table 1 in Section A.2 of this appendix to understand the relationship among these metrics.

A.2 The HTTP Server Metrics. Figure 3 shows the structure of the HTTP Server metrics and the tables below it describe the metrics which appear in the figure.

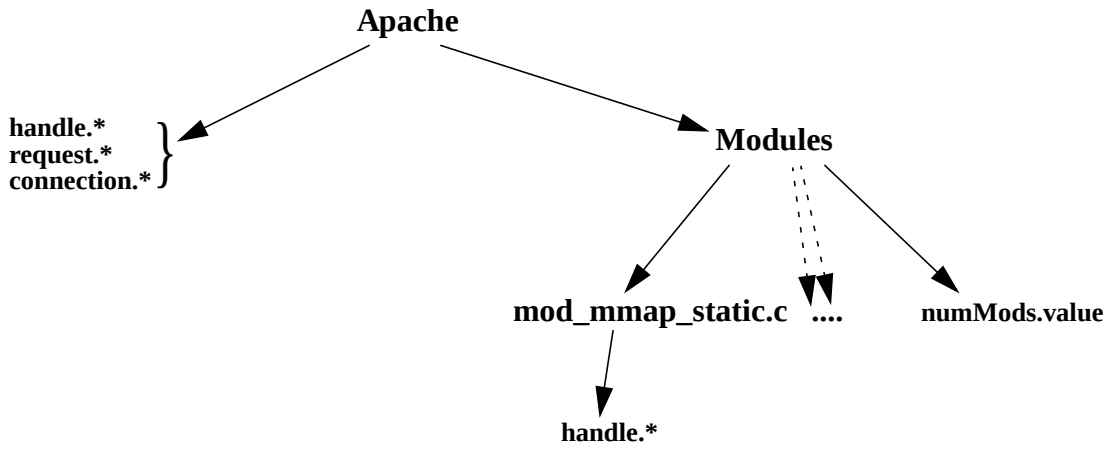

FIGURE 3. The HTTP Server Metric Tree

TABLE 1. The HTTP Server Metrics

Metric	Description	Unit
handle.maxTime	Maximum time spent in module handler	usecs
handle.minTime	Minimum time spent in module handler	usecs
handle.avg	Average time spent in module handler	usecs
handle.active	Threads currently in the handle processing phase.	threads
handle.time	Total time spent in module handler	usecs
handle.completed	Number of times the handle processing phase has completed.	ops

TABLE 1. The HTTP Server Metrics

Metric	Description	Unit
request.maxTime	Maximum time required to service an HTTP request.	usecs
request.minTime	Minimum time required to service an HTTP request.	usecs
request.avg	Average time required to service an HTTP request.	usecs
request.active	Threads currently in the request processing phase.	threads
request.time	Total time required to service an HTTP request.	usecs
request.completed	Number of times the request processing phase has completed.	ops
connection.maxTime	Maximum time spent servicing any HTTP connection	usecs
connection.minTime	Minimum time spent servicing any HTTP connection	usecs
connection.avg	Average time spent servicing HTTP connections	usecs
connection.active	Number of connections currently open	threads
connection.time	Total time spent servicing HTTP connections	usecs

TABLE 2. The Apache/Modules Metrics

Metric	Description
numMods.value	Number of loaded modules.

TABLE 3. The Apache/Modules/mod_*.c Metrics

Metric	Description	Unit
handle.maxTime	Maximum time required for this module handler.	usecs
handle.minTime	Minimum time required for this module handler.	usecs
handle.avg	Average time required for this module handler.	usecs
handle.active	Threads currently in the handle processing phase.	threads
handle.time	Total time required for this module handler.	usecs
handle.completed	Number of times the handle processing phase has completed.	ops

A.3 The JServ Metrics.

Figure 4 shows the structure of the JServ metrics and the tables below it describe the metrics which appear in each part of the tree.

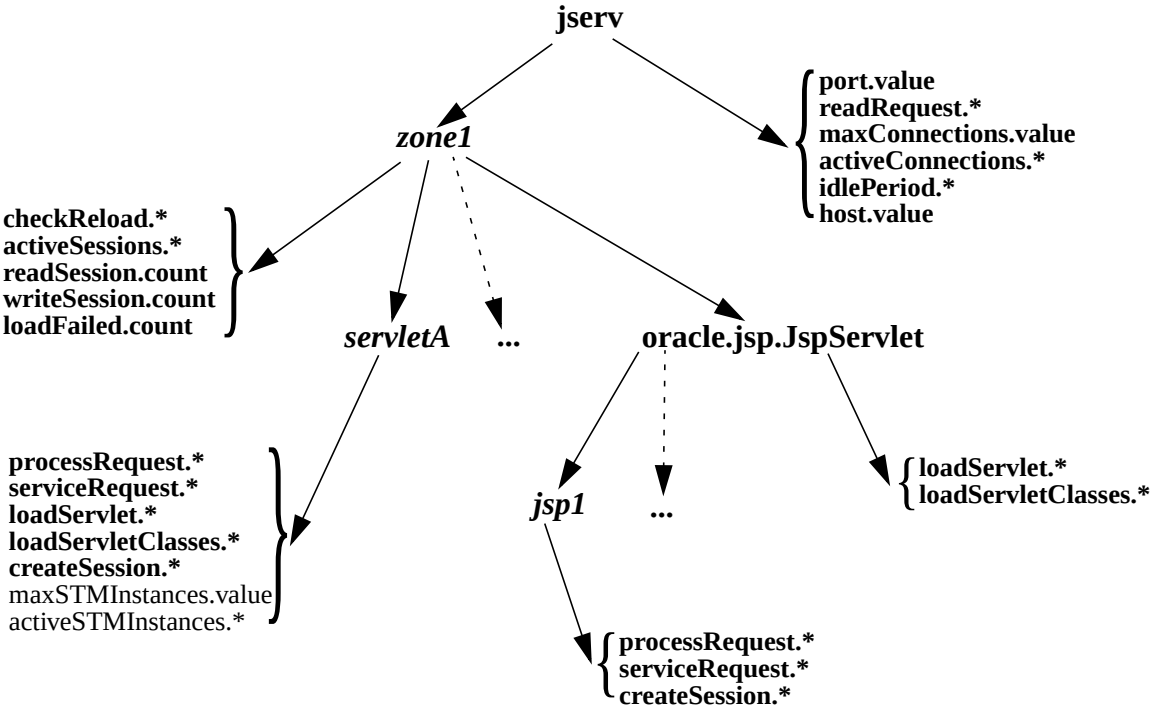

FIGURE 4. The JServ Metric Tree. Items in italics are holders for zone, servlet and jsp names. There may be more than one of each. The STM metrics only appear for servlets which implement the SingleThreadModel interface.

TABLE 4. The jserv Metrics

Metric	Description	Unit
port.value	The ID of the TCP port on which this JServ listens.	
readRequest.active	Threads currently in the readRequest processing phase.	
readRequest.avg	Average time to read and parse the incoming apjv12 request.	msecs
readRequest.maxTime	Maximum time to read and parse the incoming apjv12 request.	msecs
readRequest.minTime	Minimum time to read and parse the incoming apjv12 request.	msecs
readRequest.completed	Number of times the readRequest processing phase has completed.	ops
readRequest.time	Total time to read and parse the incoming apjv12 request.	msecs
maxConnections.value	Number of requests that can be handled concurrently in this jserv process.	threads
activeConnections.maxValue	Maximum number of requests being processed simultaneously.	threads
activeConnections.value	Number of requests being processed simultaneously.	threads
idlePeriod.maxTime	Maximum time process was not handling any requests.	msecs
idlePeriod.minTime	Minimum time process was not handling any requests.	msecs
idlePeriod.completed	Number of times no requests were being serviced.	ops
idlePeriod.time	Total time process was not handling any requests.	msecs

TABLE 4. The jserv Metrics

Metric	Description	Unit
host.value	Hostname/IP Address this Jserv process binds to.	
maxBacklog.value	Maximum number of backlog requests that may be queued in the OS waiting for this JServ	

TABLE 5. The jserv/<zone> Metrics

Metric	Description	Unit
checkReload.active	Threads currently in the checkReload processing phase.	
checkReload.avg	Average time to check if the zone must be reloaded.	msecs
checkReload.maxTime	Maximum time to check if the zone must be reloaded.	msecs
checkReload.minTime	Minimum time to check if the zone must be reloaded.	msecs
checkReload.completed	Number of times the checkReload processing phase has completed.	ops
checkReload.time	Total time to check if the zone must be reloaded.	msecs
activeSessions.value	The number of sessions which exist in this zone.	sessions
readSession.count	Number of times session data has been read with HttpSession.getValue() in this zone.	ops
writeSession.count	Number of times session data has been written with HttpSession.putValue() in this zone.	ops
loadFailed.count	Number of times we failed to load the requested application (does not work for OJSPs).	ops

TABLE 6. The /jserv/<zone>/<servlet> Metrics (Also reported in /jserv/<zone>/ oracle.jsp.JspServlet and /jserv/<zone>/oracle.jsp.JspServlet/<jsp>, see figure 4)

Metric	Description	Unit
processRequest.active	Threads currently in the processRequest processing phase.	
processRequest.avg	Average time to completely process servlet (including JServ engine overhead).	msecs
processRequest.maxTime	Maximum time to completely process servlet (including JServ engine overhead).	msecs
processRequest.minTime	Minimum time to completely process servlet (including JServ engine overhead).	msecs
processRequest.completed	Number of times the processRequest processing phase has completed.	ops
processRequest.time	Total time to completely process servlet (including JServ engine overhead).	msecs
serviceRequest.active	Threads currently in the serviceRequest processing phase.	
serviceRequest.avg	Average time for service method implementing this application (excluding JServ engine overhead).	msecs
serviceRequest.maxTime	Maximum time for service method implementing this application (excluding JServ engine overhead).	msecs
serviceRequest.minTime	Minimum time for service method implementing this application (excluding JServ engine overhead).	msecs

TABLE 6. The /jserv/<zone>/<servlet> Metrics (Also reported in /jserv/<zone>/ oracle.jsp.JspServlet and /jserv/<zone>/oracle.jsp.JspServlet/<jsp>, see figure 4)

Metric	Description	Unit
serviceRequest.completed	Number of times the serviceRequest processing phase has completed.	ops
serviceRequest.time	Total time for service method implementing this application (excluding JServ engine overhead).	msecs
loadServlet.avg	Average time to load servlet (from cache or file).	msecs
loadServlet.maxTime	Maximum time to load servlet (from cache or file).	msecs
loadServlet.minTime	Minimum time to load servlet (from cache or file).	msecs
loadServlet.completed	Number of times the loadServlet processing phase has completed.	ops
loadServlet.time	Total time to load servlet (from cache or file).	msecs
loadServletClasses.active	Threads currently in the loadServletClasses processing phase.	
loadServletClasses.avg	Average time to load servlet classes from file.	msecs
loadServletClasses.maxTime	Maximum time to load servlet classes from file.	msecs
loadServletClasses.minTime	Minimum time to load servlet classes from file.	msecs
loadServletClasses.completed	Number of times the loadServletClasses processing phase has completed.	ops
loadServletClasses.time	Total time to load servlet classes from file.	msecs
loadServlet.avg	Average time to load servlet (from cache or file).	msecs
createSession.active	Threads currently in the createSession processing phase.	
createSession.avg	Average time to create a session.	msecs
createSession.maxTime	Maximum time to create a session.	msecs
createSession.minTime	Minimum time to create a session.	msecs
createSession.completed	Number of times the createSession processing phase has completed.	ops
createSession.time	Total time to create a session.	msecs
maxSTMInstances.value	Total number of instances available for this SingleThreadModel servlet.	instances
activeSTMInstances.maxValue	Maximum number of instances concurrently servicing requests for this SingleThreadModel servlet.	instances
activeSTMInstances.value	Number of instances concurrently servicing requests for this SingleThreadModel servlet.	instances

Appendix B: Configuration.

9iAS configures DMS for you, and you probably need no further adjustments. However, a bug in 1.0.2.2 causes dmsGrab to miss metrics for some servers. Also, if your site configures JServ manually, then it must configure DMS manually too.

B.1 9iAS 1.0.2.2 Configuration Bug Workaround.

The DMS services assume that all JServs include a /root zone. This is usually configured by default but sometimes (as with the SOAP JServ server in 1.0.2.2), the root zone is omitted. You should ensure that your

JServs include a /root zone by checking the properties file for each JServ. Each properties file should contain a line that declares the root zone. This declaration looks like this:

```
zones=root
```

The root zone must declare its properties, usually with a line like this:

```
root.properties=ORACLE_HOME/Apache/Jserv/etc/zone.properties
```

ORACLE_HOME indicates the current setting of your site's ORACLE HOME directory.

B.2 Manual JServ Configuration.

If your site uses *ApJServManual on* in its jserv.conf then you need to manually configure DMS. Manual mode configuration of DMS is very easy and requires only a few additions to the jserv.conf file.

For example, here are some typical jserv.conf configuration directives for a site running two JServs, one on port 9001 and one on port 9002 of the local host (IP address 127.0.0.1):

```
ApJServHost JServ1 ajpv12://127.0.0.1:9001
ApJServHost JServ2 ajpv12://127.0.0.1:9002
```

Usually, the *ApJServHost* directives are part of a larger set of directives that configure features such as load balancing. To these directives, you must add /dms paths so that DMS clients (such as dmsGrab) can find JServ1 and JServ2 on ports 9001 and 9002. Do it like this:

```
ApJServMount /dms1 ajpv12://127.0.0.1:9001/root
ApJServMount /dms2 ajpv12://127.0.0.1:9002/root
```

The digits in the /dms... field are important. Each /dms path must be given a unique number in sequence starting with 1 (/dms0 is reserved for HTTP Server stats). Whenever you add a new JServ to your configuration, you must add a new /dms path with the next number in the sequence. Map each new /dms path to the port ID of the new JServ. In our example, /dms1 maps to port 9001 and /dms2 maps to port 9002.

Extra /dms paths won't hurt anything. If you have extra /dms paths for JServs that you are not currently using then DMS will ignore them.

Appendix C: Tools for Extracting DMS Metrics.

Oracle 9iAS version 1.0.2.2 provides no GUI for viewing performance statistics, however the does include some primitive tools for viewing the data.

C.1.1 Using the Spy Service.

The built-in Spy service can be used to gather statistics from either HTTP Server o JServ. To access the DMS data using the Spy, you need only have access to a web browser.

Once you have installed your 9iAS 1.0.2.2 server, and you are running the HTTP Server, you can access statistics about the HTTP daemon processes with the following URL:

```
http://myhost:myport/dms0/Spy
```

Note. You should replace **`myhost:myport`** with the hostname and port for accessing your webserver.

This above URL will output text similar to the following:

```
<DMSDUMP version='2.0' timestamp='995074161 (Friday, 13-Jul-2001 18:30:49
PDT)' id='25330' name='Apache'>
<statistics>
.
.
```

```
.
</statistics>
</DMSDUMP>
```

The default output format for the **Spy** is plain-text (sometimes called “raw”). To get an XML format dump of the HTTP Server statistics, modify the above URL to:

```
http://myhost:myport/dms0/Spy?format=xml
```

You should see statistics for each of the HTTP Server modules on your system, as well as summary statistics which have been gathered from all of the HTTP daemon processes running on your host. The interpretation of the HTTP Server statistics is covered in Section 2.1, and the parameters in the URL in C.1.2 below.

To access JServ statistics, we replace “dms0” with “servlet” in the URL, like this:

```
http://myhost:myport/servlet/Spy
```

In this case, you will be presented with a set of statistics for each servlet or JSP request handled by the server. You will also be presented with a set of statistics for each zone which contains a servlet that has been called. For example, since the **Spy** servlet is available in the **root** zone by default, this URL will result in statistics for both the **root** zone and for the **Spy** servlet. See Section 2.2 for a description of the JServ statistics and the relationships between them.

Note. The only thing that changed in the request to obtain JServ statistics rather than HTTP Server statistics was to specify **servlet** rather than **dms0** in the request. This is due to the default configuration provided in the HTTP Server and Jserv configuration files in iAS. If you are running your application server with multiple Jserv processes, then you may not be able to retrieve metrics from all JServs easily with your browser. In this case you should use flexmon or dmsGrab as described in the following sections.

C.1.2 Spy Arguments.

In the URLs above, the **Spy** was called with query arguments. This section describes the arguments supported by the **Spy** and how each will impact the response.

name (default: ‘/’)

As can be seen in Figures 3 and 4 in Appendix A, DMS data is organized as a tree, much like a file system. You can set the name parameter to be any valid path name known to the DMS process. In the examples above, we set **name=/**. In the data returned we see many more names. In some cases we may want to restrict the data extracted from DMS to a specific metric or set of metrics. Suppose, for example, that we only want the JServ metrics for the IsItWorking demo servlet delivered with iAS 1.0.2.2, then we would use the request:

```
http://myhost:myport/servlet/Spy?name=/jserv/root/IsItWorking
```

In iAS, the JServ metrics are organized first under the name **jserv**, then under the zone name (**root**, in the default case), and finally under the servlet name (**IsItWorking** in this case). Note that as the name is specified more explicitly, the set of metrics presented is significantly reduced.

recurse (default: **recurse=all**)

recurse=all show all known descendants under the node represented by the name parameter.
recurse=children show only this node and the direct descendants of this node.
recurse=none show only this node in the tree.

In all cases, the ancestors of the specified name will also be displayed. (To understand what this parameter does, try each of these settings with name=/jserv/root.)

format (default: **format=raw**)

format=xml produce the output in XML, often handy for further processing.
format=raw output simple text format.

units (default: **units=true**)

This argument controls whether the units (**msecs, ops,** etc.) will be shown in addition to the values.

description (default: **description=false**)

This controls whether to present the description for a family of metrics. This is only valid with **format=xml**. Metrics are sometimes grouped, producing values like the maximum and minimum for a metric that measures time. There is only one description for the entire group, though it is printed for every metric. See Appendix A for a better understanding of DMS metric groups.

value (default: value=true)

This argument controls whether to extract the value of the metrics. There are some situations in which you may need to know the names of the metrics but not their values.

C.2 flexmon.

Another tool that can be used for gathering DMS statistics is **flexmon** (available in the **dms2Client.jar** file in iAS). This tool can be used for monitoring one or more metrics over time. Suppose for example, that you want to monitor how many requests are being processed concurrently in your JServ process as time passes. Using flexmon, you can request a periodic snapshot of the **activeConnections.value**, and any other metrics you may be interested in.

C.2.1 Using flexmon.

You must first get the full address for the statistic which is of interest to you. You can get a full listing of the metrics currently published on an iAS server using

```
java flexmon -a myhost:myport -list
```

The **-a myhost:myport** argument indicates that the set of processes you want information from are to be found through port **myport** on the host **myhost**. The **-list** indicates that you want a list of the names of the metrics collected on the specified processes.

In the list of metrics, you'll see something like this:

```
/myhost/JServ:1/jserv/activeConnections.value
```

This is the name you should use to request monitoring of that metric. For example, to monitor the number of active connections on the JServ process listening on port 8007 over a period of time, you could use:

```
java flexmon -a myhost:myport /myhost/JServ:1/jserv/
activeConnections.value
```

This will result in a report of the concurrent number of requests in JServ:1. The value will be updated once per second (the default). Because **flexmon** makes use of the **Spy** to get the data, you'll notice the value for this metric will always be at least 1, as a connection must be active in order to service your request.

C.2.2 flexmon Arguments.

Using flexmon, you can monitor your system in a variety of ways by working with the arguments described below. The syntax for flexmon is

```
java flexmon -address <host[:port]> -list

java flexmon -address <host[:port]>
```

```
                              [-interval <secs>] [-count <num>] <metric> <metric>
...
```

In the first case, **flexmon** will provide a list of all metrics available for the given address. In the second case, it is used to monitor the specified metrics. The metric names used as input for monitoring in the second case must be obtained from the list produced using the first case. We see an example of this in Section 2.2.1 above.

The arguments to flexmon are:

```
-a[ddress] <host[:port]>
```

host is the domain name or IP address of the host on which the HTTP Server is running. The default is localhost.

port is the port on which the HTTP Server is listening. The default is 80.

If your site is configured with "ApJServManual auto" (usually set in jserv.conf), then flexmon will be able to find metrics for the HTTP Server and all of your JServs automatically.

```
-l[ist]
```

produce a list of all metrics available for the host, port and numTargets combination specified with -address.

```
-i[nterval] <secs>
```

number of seconds to wait between metric retrievals (not used with the -list option). The default is 1 second.

```
-c[ount] <num>
```

specifies number of times to retrieve values when monitoring metrics (not used with the -list option). The default is infinity (e.g. continue retrieving metric values until the process is stopped).

Example. Suppose you are balancing the load on your host across four JServ processes and that you want to monitor the number of requests handled by each over time. Then you can generate a list of the available metrics, and grep for what you want to monitor on the JServ processes:

```
java flexmon -a myhost:myport -list | \
grep "IsItWorking/processRequest.completed"
```

This will result in output such as:

```
/myhost/JServ:9001/jserv/root/IsItWorking/processRequest.completed
/myhost/JServ:9002/jserv/root/IsItWorking/processRequest.completed
/myhost/JServ:9003/jserv/root/IsItWorking/processRequest.completed
/myhost/JServ:9004/jserv/root/IsItWorking/processRequest.completed
```

To monitor the load balancing for the IsItWorking servlet among JServ processes for two hours, you could use:

```
java flexmon -a myhost:myport:5 -i 60 -c 120 \
/myhost/JServ:9001/jserv/root/IsItWorking/processRequest.completed \
/myhost/JServ:9002/jserv/root/IsItWorking/processRequest.completed \
/myhost/JServ:9003/jserv/root/IsItWorking/processRequest.completed \
/myhost/JServ:9004/jserv/root/IsItWorking/processRequest.completed
```

Depending on your load, this will result in output such as

```
Mon Apr 09 17:13:01 PDT 2001
/myhost/JServ:9001/jserv/root/IsItWorking/processRequest.completed  437 ops
/myhost/JServ:9002/jserv/root/IsItWorking/processRequest.completed  441 ops
```

```
/myhost/JServ:9003/jserv/root/IsItWorking/processRequest.completed  432 ops
/myhost/JServ:9004/jserv/root/IsItWorking/processRequest.completed  436 ops
Mon Apr 09 17:13:11 PDT 2001
/myhost/JServ:9001/jserv/root/IsItWorking/processRequest.completed  489 ops
/myhost/JServ:9002/jserv/root/IsItWorking/processRequest.completed  490 ops
/myhost/JServ:9003/jserv/root/IsItWorking/processRequest.completed  476 ops
/myhost/JServ:9004/jserv/root/IsItWorking/processRequest.completed  481 ops
...
```

It goes without saying developing little scripts for monitoring specific metrics over time is the most popular way to use flexmon.

References.

1. Oracle 9i Application Server Oracle HTTP Server Performance Guide