How to Tune an Environment - 9.1 and Beyond

Introduction

A well-tuned environment that uses the appropriate system resources (not too little, not too much) is critical to ensuring a happy user and a happier customer. Unfortunately, today most people consider the task of tuning an environment akin to sprinkling some magic dust or turning some dials while praying to a higher power.

This guide discusses the elements involved in tuning an environment, goes into detail what needs to be taken into account during tuning and provides a starting point for the most often-tuned parameters to help ease the pain involved in getting an environment to be performant without consuming too many resources.

In order for the tuning of an environment to ultimately be successful it needs to be looked at as an iterative process. A "set it and forget it" approach will rarely result in a well-tuned environment. Read through this entire document, follow its suggestions, monitor the environment for a while, tweak values where appropriate and repeat a few times. And don't forget that adding a new component, applying a hot fix, or even adding a number of new locations to a warehouse are all examples of things that may require further tuning of an environment as time goes on.

It's assumed that you have a general understand of the MOCA architecture before getting started. If terms like "MOCA Server", "native process" or "database connection pool" are foreign to you then you will need to gain a better understanding of what MOCA is and how it's architected first. You wouldn't change the oil on your car if you didn't know what an oil filter is so you shouldn't be setting the native process pool's maximum size if you don't know what a native process is.

Getting Started

Before turning all the knobs and flipping all the switches it's important to take a step back and understand how the environment will be used by the customer as no two environments are the same. Is the environment being used by a large number of users? Are there custom integrations that require a process-based tasks? What platform is the environment running on and how much memory is available to the environment? Those questions are just a few that you need to have answers to *before* starting to tune an environment. If you don't have them yet you probably shouldn't be trying to tune the environment yet either.

Operating System

Tuning the OS is generally outside the scope of this document due to the number of supported platforms and differences between them. It is, however, important to ensure that any settings won't be constrained by the OS or shell. To that end we've highlighted several critical OS configurations that are sometimes overlooked by system administrators.

Linux/UNIX

ulimit

If you need to set the max heap for the MOCA Server to 8GB you need to make sure on Linux/UNIX systems that the kernel doesn't limit the amount of memory a process can allocate and that the shell's "data size" set via the `ulimit` command will allow it.

Problems most often arise on Linux/UNIX platforms when a `ulimit` value is too restricting. The following common `ulimit` values are suggested as a starting point on Linux/UNIX platforms:

```
-f   file size(blocks)         unlimited
-d   data seg size(kbytes)     unlimited
-s   stack size(kbytes)        10240
-m   memory size(kbytes)       unlimited
-u   max user processes        4096
-n   open files                4096
```

urandom

MOCA relies on Java's `SecureRandom` library to provide randomized seed values for various security tokens and hashes. The Oracle JVM by default is configured on the Linux/UNIX platform to pull randomization data from `/dev/random`. For some customers the default `/dev/random` implementation may perform too slowly and lead to unexpected delays in the application. These customers are encouraged to tune their OS' `/dev/random` implementation (outside of the scope of this document) or switch to the less secure but faster `/dev/urandom` (see Configuring Java's Secure Random Number Generation).

Registry Values

The majority of the work required to tune an environment involves changing registry key values in the registry file. The Developer Guide documents every registry key. This guide includes more detailed information on the registry keys that are more likely to require changing in order to tune an environment. Each section below equates to a section of the registry file. (e.g. Java, Server, etc...) Each section then includes information about the different registry keys that may need changing.

Java Section

`java.vmargs`

The `java.vmargs` registry key value defines the arguments that will be passed to the "default" Java VM on start up. The default JVM is the JVM that is used by the MOCA Server and process-based tasks whose command line references the JAVA environment variable.

The `java.vmargs` registry key value must include options that are valid for the Java application launcher included in the Java distribution being used. JDA certifies against the Oracle HotSpot Server VM on the Windows and Linux platforms, the IBM Virtual Machine for Java on AIX platforms and the HP-UX HotSpot Server VM on HP-UX platforms. The following web pages provide detailed information on valid options for each:

- AIX
- HP-UX *HP supports the Oracle HotSpot Server JVM's options.*
- Linux
- Windows

There are a few options that are commonly used and deserve extra attention:

`-Xms`

Specifies the initial size of the object heap (or memory allocation pool). This value consists of a numeric size followed by a unit of measure (m|M for megabyte, g|G for gigabyte). The default value is chosen at runtime based on system configuration. The -Xms value should not exceed the value set for -Xmx For more information, see HotSpot Ergonomics.

Format: `-Xms<size>[g|G|m|M]`

Examples: `-Xms64m`

`-Xmx`

Specifies the maximum size of the object heap (or memory allocation pool). This value consists of a numeric size followed by a unit of measure (m|M for megabyte, g|G for gigabyte). The default value is chosen at runtime based on system configuration (usually 1/4 of system memory or 1G, whichever is larger). For more information, see HotSpot Ergonomics.

Format: `-Xmx<size>[g|G|m|M]`

Examples: `-Xmx128m`

The MOCA Server is a multi-threaded Java process that manages client requests, jobs, tasks and other core elements. When client requests are received the MOCA Server interprets the request as a MOCA command, executes the component associated with that command and returns the results to the caller. The implication of this is that the MOCA Server needs enough memory available to it in order to support every simultaneous request and subsequent result set in addition to the other intrinsic overhead associated with the MOCA Server. As the MOCA Server approaches the limit of its allocated memory performance may suffer as the frequency of garbage collection increases. If the MOCA Server runs out of memory the JVM will throw an `OutOfMemoryError` and crash. Therefore, it's critical to define a max heap size via Java's `-Xmx` command line option appropriately via the `java.vmargs` registry key.

Since the addition of support for Java-based components in MOCA the most often asked question by someone trying to determine what to set `java.vmargs` to may well be "what should my max heap be set to?" Unfortunately, there is no simple answer to this. That's not because we're too lazy to figure it out. It's because every installation is slightly different so a setting that may be optimal for one customer may have disastrous consequences for another. So it's critical to monitor and revisit this registry key value until the environment is fully tuned.

Registry key:	`java.vmargs`
Default value:	Determined by the JVM
Suggested starting value:	`-Xmx4096m`

The console's Resource Usage page provides a graph that shows the current, peak and max defined memory. The goal is to have the peak value around 60% of the max value.

MOCA Servers that crash due to an `OutOfMemoryError` should be configured with a larger max heap size.

java.native-vmargs

The java.native-vmargs registry key value defines the arguments that will be passed to a JVM that is used by native processes and C-based applications that call osLaunchJavaApplication().

When optimizing your native process Java configuration please be mindful of the following considerations:

1. **Java Heap Size**
   Native processes execute native (C and C++) code. As such, the amount of system memory they use is not governed by the Java heap size arguments (-Xms and -Xmx). Therefore, it's usually not necessary to configure a large heap size for native processes; in fact it can adversely affect system stability by needlessly reserving system resources that could be used elsewhere. We recommend configuring your -Xms with 128m as a starting point. If a native process crashes due to an OutOfMemoryError the number should be increased in 128m increments. If the native process crashes with osPanic() calls in the call stack of its core file the number should be decreased in 128m decrements. For more information please see the "Common Problems" section below for more information.

2. **Native Process Maximum Pool Size**
   When configuring the size of the native-vmargs Java heap it's important to keep in mind the maximum number of native processes that the system is capable of starting at any given time (controlled by the server.max-pool-size registry value which is discussed in greater detail below). The native process pool is comprised of many individual JVMs, each with their own reserved heap memory. When deciding on an appropriate Java heap size it's important to consider to total memory usage of the entire native process pool and whether the system has sufficient memory. If not, it may be necessary to lower the Java heap size (but not below 128m) or shrink the size of the native process pool.

Registry key:	java.native-vmargs
Default value:	Determined by the JVM
Suggested starting value:	-Xmx128m

java.vmargs.application_name

The java.vmargs.*application_name* registry key value defines the arguments that will be passed to the Java VM for applications that call osLaunchJavaApplication() with a second argument of *application_name*. The following MOCA applications call osLaunchJavaApplication(); therefore, they support java.vmargs.*application_name*:

createctl	mgendoc	mocaserver
dbupgrade	mload	runtask
installsql	mloadall	

There is no default value for java.vmargs.*application_name* as this is completely dependent on the requirements of the actual application. Applications that are deployed as part of the standard product that require a special value should already be defining that special value so there should be no need to modify any of these values.

Server Section

server.min-idle-pool-size

The server.min-idle-pool-size registry key defines the minimum number of idle native processes that will be maintained in the pool. If the total number of idle and busy processes equals the max-pool-size value, then another process will not be spawned. Setting the value of min-idle-pool-size to 0 will cause processes to only be created as requested. Setting this value equal to the max-pool-size value will ensure that the pool always has that many processes in it.

Registry key:	server.min-idle-pool-size
Default value:	1
Suggested starting value:	10% of max-pool-size

The suggested starting value is an estimate that can be tweaked safely. Native processes are expensive, therefore we don't want to create too many if they're not being utilized. 10% is a safe number to start with once you've determined how many max pool size you will need. If you determine that you're waiting on native process creations, you can increase the number of idle native processes.

server.max-pool-size

The `server.max-pool-size` registry key value defines the maximum number of native processes that can exist in the native process pool at a time.

Components implemented in C or C++ execute within a native process. When a request is made to execute a C or C++ component a thread within the MOCA Server requests a native process from the native process pool to handle execution of the C or C++ code. If every native process is already in use a new native process will be spawned if `server.max-pool-size` has not already been reached. If `server.max-pool-size` native processes already exist the request will block waiting for a native process to become available. A timeout occurs after waiting `server.process-timeout` seconds for a native process to become available.

Registry key:	`server.max-pool-size`
Default value:	20
Suggested starting value:	20

The suggested starting value above is a very rough estimate. The console's Resource Usage page provides a graph that shows the current, peak and max native process pool size. This should be monitored closely during tuning. The goal is to have the peak value around 70% of the max value. A warning message is also written if a native process isn't immediately available for a request. Environments showing a peak value greater than 70% with trace files showing the message "Waiting for a native process to become available" should consider increasing the `server.max-pool-size` registry key value.

server.classic-pool-size

The `server.classic-pool-size` registry key value defines the number of threads to be used for executing incoming socket requests on the classic protocol listener. The 2010.1 and newer releases employ an HTTP-based protocol, which is not compatible with older releases. A classic listener can be enabled by defining a `server.classic-port` registry key value to allow older versions to issue remote requests to newer versions. This value only takes effect if the `server.classic-port` registry key value is set.

Registry key:	`server.classic-pool-size`
Default value:	20
Suggested starting value:	20

server.process-timeout

The `server.process-timeout` registry key value defines how long the server will wait (in seconds) for a native process to start up before throwing an error.

Registry key:	`server.process-timeout`
Default value:	20
Suggested starting value:	20

A warning message is written if a native process takes longer than `server.process-timeout` to start up. Trace files showing the message "A timeout occurred attempting to spawn a native process" can be indicative of an overwhelmed system and should be investigated further.

server.query-limit

The `server.query-limit` registry key value defines the maximum number of rows a query can return. If a SQL query returns more than `server.query-limit` rows, an exception is thrown and the transaction rolls back. This value van be set to zero (0) for no limit.

Registry key:	`server.query-limit`
Default value:	100000
Suggested starting value:	100000

A warning message is written when a query is executed that attempts to exceed the `server.query-limit` registry key value. Trace files showing the message "Maximum rows exceeded in query" can be indicative of either a misbehaving application or component or a need to increase the `server.query-limit` registry key value. However, the `server.query-limit` registry key value should *not* be blindly increased without first ensuring that there are no misbehaving applications or components. Doing so could adversely effect the overall system performance.

server.max-commands

The `server.max-commands` registry key value defines after how many requests a native process will be shutdown.

Registry key:	`server.max-commands`
Default value:	10000
Suggested starting value:	Once every 24 hours

In theory there is no real reason to shutdown a native process. It should be able to run for the life of the MOCA Server. In practice, however, C-based components that leak memory can cause the memory footprint of a native process to grow over time to the point that it can effect other processes running on the box and eventually crash when it can't allocate any more memory. The suggested starting value for shutting down a native process should be determined by looking at how many commands execute within a 24 period and setting the value using that number.

`server.max-async-threads`

The `server.max-async-threads` registry key value defines the maximum number of asynchronous threads that are available to execute asynchronous execution tasks.

Registry key:	`server.max-async-threads`
Default value:	20
Suggested starting value:	20

The console's Resource Usage page provides a graph that shows the current, peak and max number of asynchronous threads. The goal is to have the peak value around 75% of the max value.

`server.compression`

The `server.compression` registry key value defines a flag specifying if the server response is compressed. Setting this to true enables compression of the server response.

The MOCA Server supports the compression of results prior to sending them back to the client. Additional time is spent by the MOCA Server compressing those results, consuming CPU on the application server. Most networks are fast enough that compression of results doesn't provide any added value. However, significant performance improvements have been seen on slow networks by enabling compression.

Registry key:	`server.compression`
Default value:	false
Suggested starting value:	false

Database Section

`database.min-idle-conn`

The `database.min-idle-conn` registry key value defines the minimum number of idle database connections that will be maintained in the pool. If the total number of idle and busy connections equals the max-conn value, then another connection will not be created. Setting the value of min-idle-conn to 0 will cause connections to only be created as requested. Setting this value equal to the max-conn value will ensure that the pool always has that many connections in it.

Registry key:	`database.min-idle-conn`
Default value:	1
Suggested starting value:	Start with 10% of max-conn setting. If wait times occur, increase by 5%

The suggested starting value is an estimate that can be tweaked safely. Database connections can be expensive, therefore we don't want to create too many if they're not being utilized. 10% is a safe number to start with once you've determined how many max connections you will need.

`database.max-conn`

The `database.max-conn` registry key value defines the maximum number of database connections that can exist in the database connection pool at a time.

When a request is made for a database connection the database connection pool is searched for a database connection that isn't already being used. If every database connection is already in use a new database connection will be established if `database.max-conn` has not already been reached. If `database.max-conn` database connections already exist the request will block waiting for a database connection to become available. A timeout occurs after waiting `database.conn-timeout` seconds for a native process to become available.

Registry key:	`database.max-conn`
Default value:	100
Suggested starting value:	The greater of 100 or the number of users + The number of thread-based tasks

The suggested starting value above is a very rough estimate. The console's Resource Usage page provides a graph that shows the current, peak and max database connection pool size. This should be monitored closely during tuning. The goal is to have the peak value around 70% of the max value.

`database.conn-timeout`

The `database.conn-timeout` registry key value defines the number of seconds to wait before timing out attempts to get a connection from the database connection pool.

Registry key:	`database.conn-timeout`
Default value:	30
Suggested starting value:	30

A warning message is written if a database connection timeouts occurs. Trace files showing the message "A timeout occurred attempting to get a database connection from the pool" should consider increasing the `database.max-conn` registry key value after determining that the timeout didn't occur due to an abundance of database locks or transactions that are running too long. This message should always be found in tandem with the max connection message discussed above.

`database.login-timeout`

The `database.login-timeout` registry key value defines the number of seconds to wait before timing out attempts to establish a new database connection.

Registry key:	`database.login-timeout`
Default value:	30
Suggested starting value:	30

A warning message is written if a database login timeouts occurs. Trace files showing the message "A timeout occurred attempting to establish a new database connection" can be indicative of an overwhelmed database engine or network connectivity issue and should be investigated further.

Monitoring Section

`monitoring.csv-reporter-keep-hours`

The `monitoring.csv-reporter-keep-hours` registry key value defines the maximum number of hours worth of M&D probe data the system will retain in the CSV data files (default location `$LESDIR/data/csv_probe_data`). The M&D probes are a valuable source of system performance and resource utilization information. If the system is experiencing sluggish performance or you're contemplating a change to system resources you should consider increasing the amount of probe data retained. With more data you can create a more accurate representation of the system's behavior and make more informed tuning and configuration choices.

Registry key:	`monitoring.csv-reporter-keep-hours`
Default value:	72
Suggested starting value:	72

Jetty

Jetty is the web server container that handles incoming requests to MOCA. This section describes a few of the configuration options that Jetty provides.

Where is Jetty configured?

Jetty is configurable through a file that must be named `jetty.xml` (ignoring case) that is found by searching the directories defined by the `server.prod-dirs` registry key value. The first `jetty.xml` file found is used. This implies that a `jetty.xml` file should only exist in one directory defined by the `server.prod-dirs` registry key value.

Thread Pool Configuration

Jetty internally keeps a pool of threads that it reuses for various requests. If the pool is completely exhausted of threads, then additional requests will have to wait until a currently running request completes and returns the thread back to the pool.

The below snippet shows how you would configure the `ThreadPool` within Jetty.

```
<Configure class="org.eclipse.jetty.server.Server">
  <Set name="threadPool">
    <New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
      <Set name="minThreads">10</Set>
      <Set name="maxThreads">1000</Set>
    </New>
  </Set>
</Configure>
```

Note that there is both a `minThreads` and `maxThreads` attribute. The `minThreads` attribute is of less use then the `maxThreads` attribute, since threads can be spawned much faster than whole processes. Therefore, it is normally suggested to keep the `minThreads` attribute relatively low to reduce memory usage when inactive.

The default values for both of these attributes will vary on which Jetty version is in use. As of this writing the default values are 8 and 254 respectively.

Additional Information

The Jetty website has more detailed information on these parameters and others that can be configured via the `jetty.xml` file.

- Jetty XML Syntax
- Jetty API

Memory

The goal of this section is to provide an understanding of what the memory footprint of an environment will be. The converse of this is that it's critical to have enough physical memory available on the box for an environment. There is simply no way to tune an environment on a box that doesn't have enough memory for the environment. If you get into this situation the question shouldn't be "how can we change the parameters so our environment doesn't consume more than a given amount of memory?" Instead, it should be "how much more memory do we need to add to the box for an environment?"

How much memory is available?

The application servers that the JDA SCE suite are deployed on are almost always dedicated to the SCE suite. However, opening up the cover and looking at how much memory is physically installed is *not* going to provide the right answer to this question. The OS and its associated services and processes consume memory as well and need to be taken into account. For example, on my 8GB laptop running Windows 7 the OS and its base processes consume ~2GB on an otherwise idle system. So for my 8GB laptop that leaves 6GB of memory available for the SCE suite. If I had assumed I had 8GB available I would have over-estimated my laptop's capacity by 25%.

But what about virtual memory? Can't I rely on my swap space to make up for the difference?

The short answer to this is that you probably don't want to. Imagine a box with 4GB of physical memory, 8GB of swap space and a process that is allowed to grow above that 4GB of physical memory. After consuming all the available physical memory the OS will start swapping. The same would be true if you had a number of 1GB processes running on the box. As one executes it may cause another to get swapped out. From a user's perspective both of these scenarios will cause application performance to grind to a halt and become a problem.

How much memory is my environment going to consume?

There are a number of different processes that contribute to the memory footprint of an environment. The following are the most common:

- The MOCA Server
- Native processes
- Process-based Tasks
- Services

The memory footprint of an environment is the sum of each of the above. Strategies for estimating how much memory these processes individually consume are presented below. After calculating values for each you can prepare a memory footprint estimate of the entire environment simply by summing these together.

MOCA Server

Look at the `java.vmargs` registry key value for the max heap (`-Xmx`) argument. The value of this argument defines approximately how much memory the MOCA Server will consume.

	Value	Note
Max Heap Size:		`java.vmargs`

Native Processes

Each native process uses a combination of Java heap and native memory. Look at the `java.native-vmargs` registry key value for the max heap (`-Xmx`) argument. This argument defines how much heap memory each native process can consume. Next add to it an estimated amount of native memory (memory claimed by the native C / C++ code during execution). If you don't know how much native memory your native processes are using on average we recommend starting with 100m for an estimate. Combine these two values to create an estimated memory cost per native process. Then multiply by the maximum number of native processes the system is configured to support (the value of the `server.max-pool-size` registry key) to calculate how much memory the native process pool could potentially consume.

	Value	Note
Max Pool Size:		`server.max-pool-size`
Max Heap Size:		`java.native-vmargs`
Total Native Processes:		Max Pool Size * (Max Heap Size + native memory estimate)

Process-based Tasks

Look at the max heap (`-Xmx`) argument from the command line of any process-based tasks written in Java. Process-based tasks written in a language other than Java will be more difficult to determine. The team that wrote the task may be able to provide some insight, but the best option is to monitor the size of the process associated with the task over time.

	Value	Note
Task 1:		Task's max heap setting or observed memory usage
Task 2:		Task's max heap setting or observed memory usage
Task 3:		Task's max heap setting or observed memory usage
Task 4:		Task's max heap setting or observed memory usage
Task 5:		Task's max heap setting or observed memory usage
Total Tasks:		Sum of all the above

Other Services

Look at the max heap (`-Xmx`) argument from the command line of any services written in Java. Services written in a language other than Java will be more difficult to determine. The team that wrote the service may be able to provide some insight, but the best option is to monitor the size of the process associated with the service over time.

	Value	Note
Service 1:		Service's max heap setting or observed memory usage
Service 2:		Service's max heap setting or observed memory usage
Service 3:		Service's max heap setting or observed memory usage
Total Services:		Sum of all the above

Native Memory

In this section we have focused on the Java heap because it's traditionally the largest overall consumer of memory in MOCA. However, it's important to understand that not all JVM memory is heap; the need to understand this concept has grown more important with the release of Java 8. Starting in Java 8, the *perm gen* space (class metadata and interned Strings) is no longer part of the Java heap. It now belongs to a new *Meta space* which utilizes native memory. This means it is possible to bring down an entire system if the application loads too many classes or interned strings because the maximum amount of memory consumed is no longer limited by the Java heap (`-Xmx` argument). If this is a concern you can set the new `-XX:MetaspaceSize` vmarg to limit the amount of native memory used by the metaspace. At this time we do not believe this is necessary (metaspace should be relatively small) so we do not recommend using this option.

To obtain a detailed picture of Java's memory usage we recommend using Oracle's jcmd utility. More information is available here.

Jobs and Tasks

Jobs are nothing more than commands that run on a schedule. Those commands execute within a thread of the MOCA Server. There are two flavors of tasks - thread-based and process-based. Thread-based tasks run within a thread of the MOCA Server. Process-based tasks are discrete processes that consume their own memory and have their own footprint.

The following are some guidelines to follow when working with jobs and tasks:

1. Prefer jobs over tasks.
2. Prefer thread-based tasks over process-based tasks. Process-based tasks are discrete processes and will consume more memory than the same thread-based task running within a thread of the MOCA Server. The only reason to use a process-based task would be if there were special requirements for the process. e.g. different JVM arguments
3. Distribute the schedule for jobs. If you have ten jobs that need to run once a day, rather than running all ten at the same time distribute them so that only one job will run at a time.
4. Set the max heap for process-based tasks written in Java to a reasonable value that won't consume too much memory. While setting a task's max heap to 8GB may ensure it will never run out of memory it is also wasteful and will negatively impact the memory footprint of the environment as a whole.

Command Profiling

After having general performance to a satisfactory point there are still areas that make sense to investigate for additional performance improvement opportunities. Even a single command that "does the wrong thing" can adversely impact an entire environment. Command profiling provides a way to analyze the performance of individual commands executing within the MOCA Server and identify problem commands that are running too long.

See the Command Profiling chapter of the Developer Guide for more information on how to use command profiling.

Monitoring and Diagnostic Probe Data

MOCA comes with a built-in Monitoring and Diagnostic framework (M&D). By default, the M&D probe data collected by the system is written to CSV files every five minutes (default output path is `$LESDIR/data/csv_probe_data`). When experiencing system performance slowdowns or evaluating hardware changes you should consider reviewing the probe data to better make informed choices. The base probes distributed by MOCA capture system resource usage, JVM resource usage and the usage of the core MOCA resources (database connections, native process, etc). An exhaustive list of probes distributed by MOCA is available in Monitoring and Diagnostics in MOCA portion of the MOCA Developer Guide.

Support Zip

MOCA's Support Zips contain a wealth of information about the health and stability of the system. Support Zips are generated from the MOCA Console (see also MOCA Console - Support File). In addition to M&D probe data; the Support Zip contains details about the current sessions, users, native processes, JVM threads, jobs, tasks and more. The support zip also contains a `resource-usage.txt` file which records the system's maximum resource usage since the last MOCA server restart. This information is useful when evaluating changes to the MOCA registry or hardware sizing.

resource-usage.txt

```
current_heap_used=1098.0
current_heap_size=2475.0
max_heap_size=7282.0
delta_current_heap_size=1377.0
delta_max_heap_size=4807.0
current_sessions=0
peak_sessions=7
max_sessions=10000
delta_peak_sessions=7
delta_max_sessions=9993
current_native_processes=37
peak_native_processes=37
max_native_processes=50
delta_peak_processes=0
delta_max_processes=13
current_db_connections=56
peak_db_connections=56
max_db_connections=100
delta_peak_db_connections=0
delta_max_db_connections=44
current_async_threads_used=0
current_async_threads_size=8
max_async_threads_size=20
delta_current_async_threads_size=8
delta_max_async_threads_size=12
```

Common Problems

My MOCA Server crashes with an `OutOfMemoryError`.

MOCA Servers that crash due to an `OutOfMemoryError` need to be configured with a larger max heap size via the `java.vmargs` registry key. See the MOCA Server section above for more information.

It is possible to have a memory leak in Java code so the MOCA Server should be monitored via a tool like VisualVM to ensure that the MOCA Server doesn't exhibit a continual and slow increase in the amount of memory it is consuming over time. If a memory leak is suspected a series of heap dumps should be taken over time to troubleshoot the problem.

My native processes crashes and the Java hot spot error log shows an `OutOfMemoryError`.

Native processes that crash due to an `OutOfMemoryError` need to be configured with a larger max heap size via the `java.native-vmargs r` egistry key. See the Native Processes section above for more information.

My native process crashes and the core dump shows an `osPanic()` call in the call stack.

Native processes that crash in this manner need to be configured with a *smaller* max heap size via the `java.native-vmargs` registry key. See the Native Processes section above for more information.

It is possible to have a memory leak that causes frequent native process crashes so native processes should be monitored via a tool like the Process Explorer to ensure that they don't exhibit a continual and slow increase in the amount of memory they are consuming over time. If a memory leak is suspected tools like `MOCA_DMALLOC` should be used to troubleshoot the problem.

My trace shows long periods of inactivity between MOCA commands

When observing long periods of inactivity between MOCA command executions in a trace we should first identify if the missing time falls between two client requests. If so it's likely the client is busy working or paused waiting for user input rather than a problem with the MOCA server.

Example client request gap:

```
2017-11-09 09:00:21,672 DEBUG [97  4a02] DefaultServerContext [0] Server
Context Closed []
2017-11-09 09:00:21,672 DEBUG [97  4a02] CommandDispatcher [0] Dispatched
command []
2017-11-09 09:00:22,423 DEBUG [97  4a02] CommandDispatcher [0] Dispatching
command... []
2017-11-09 09:00:22,423 DEBUG [97  4a02] CommandDispatcher [0] Server got:
set trace where activate = 0  []
```

Notice the missing time between the *Dispatched command* and *Dispatching command...* messages. It's also important to match up the thread and session ids (*[97 4a02]*) when making this comparison.

If you're confident the client is rapidly dispatching requests and not paused yet a noticeable gap remains we recommend investigating the customer's DNS configuration. To do that, configure the client to use the MOCA server's raw IP address when connecting instead of the DNS name. If the performance problem is resolved, it indicates the customer's DNS configuration is performing poorly and should be investigated.

MOCA Client Best Practices

Under normal conditions MOCA is a highly performant component architecture. However, over the years we have identified several best practices that customers should follow to get the most out of the MOCA framework.

- **Reuse MOCA connections**
  MOCA client connections (HTTP) can be reused after being authenticated. The authentication process can be expensive so it's recommended customers pool their MOCA connections to minimize authentication costs and the number of client connections opened on the MOCA server.

- **Minimize client requests**
  MOCA supports both Local Syntax and Groovy script which can be used to perform work on the server. When multiple queries or business processes must be chained together it's typically a better use of resources do so on the server wherever possible instead of returning data to the client and then dispatching additional requests.

Resources

How to Determine Maximum Heap Size