**JBoss AS 5.0 Performance Tuning and Scaling**

This document is an approach to find out methods and parameters involved in tuning an application running in a distributed environment. The document tries to capture and evaluate performance parameters involved at Application/ Web Server, Hardware, Software, Network and Database (Oracle 11g) levels.

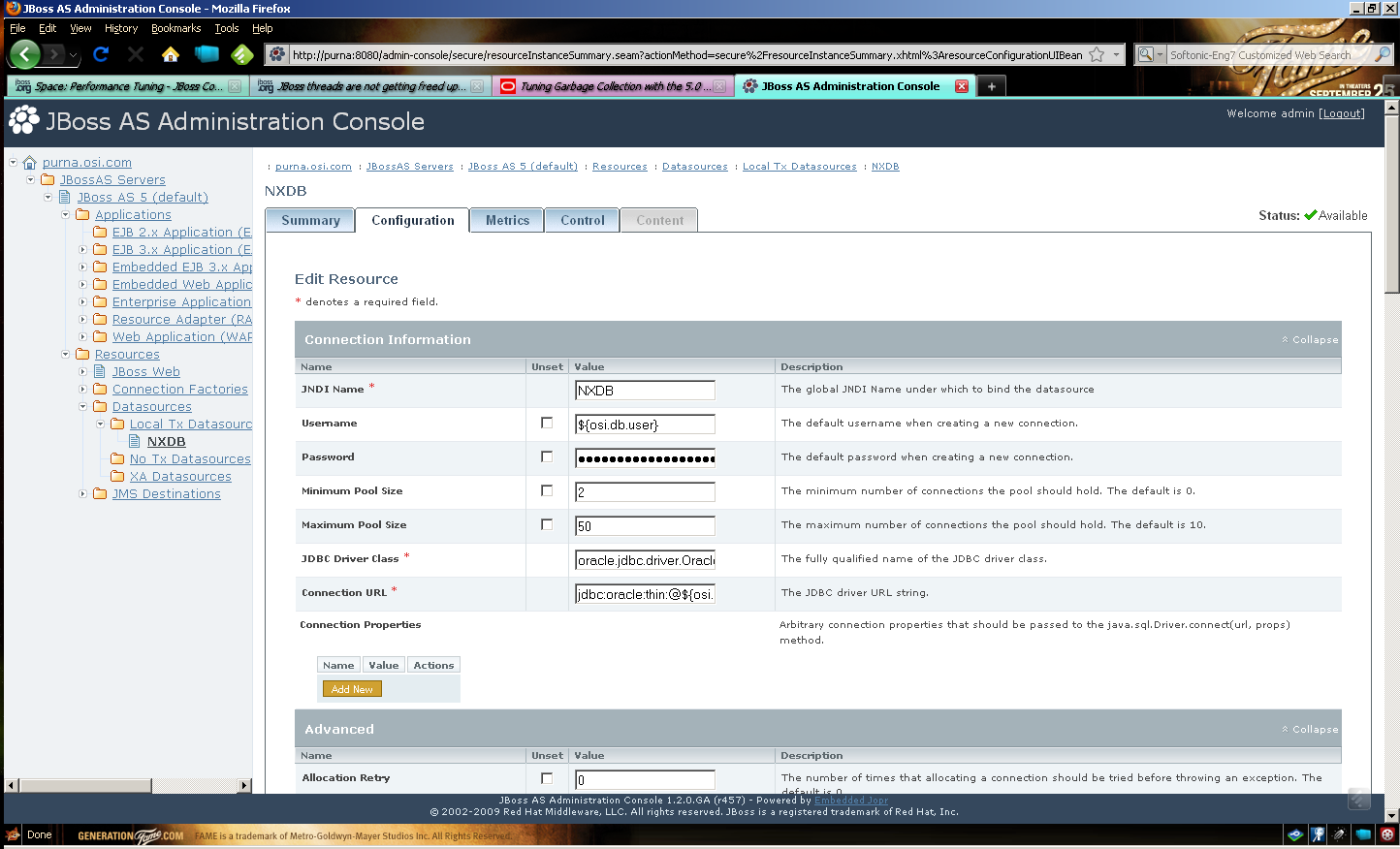
**JBoss performance tuning areas**

1. Connection Pooling
2. Thread Pooling
3. Object and component pools.
4. Logging
5. Caching
6. JVM Tuning
7. JBoss Clustering
8. Latest JDBC Drivers

**1. Connection Pooling**

Connection pooling and thread pooling are the most important areas to consider when you want to maximize throughput on modern hardware. To take advantage of the robust connection pooling in JBoss EAP, start by adjusting connection pool settings on the data source definitions that you can set up in the deploy directory. Set the minimum pool size to the level you want to tune for, and then set the maximum at least 25-30% higher. Don’t be concerned about setting the maximum too high, because if you don’t need that many connections, the pool will shrink automatically.

To determine the proper sizing, you can monitor your connection usage. A pool that is too small will throttle the application, as JBoss EAP will queue the request for a default of 30,000 milliseconds (or 30 seconds) before giving up and throwing an exception. If you start seeing a lot of 30-second timeouts, that is a strong clue that you need to look at your connection pooling. You can monitor the connection pool utilization from the EAP JMX console, from JBoss ON, or from database-specific tools. These data source connection settings can be edited through Administration Console of JBoss



**2. Thread Pooling**

JBoss EAP has robust thread pooling, but before you can size the thread pools appropriately, you need to know how they are used and which ones might be affecting your application’s performance. The characteristics of your specific application will determine which thread pools are used and which ones might become bottlenecks. This can vary significantly from application to application. The table below provides a summary of how each thread pool is used.

|  |  |  |
| --- | --- | --- |
| **Thread Pool Where is it Defined How is it Used?** | | |
| System Thread Pool | In jboss-service.xml in the conf directory | For JNDI naming — the default setting is fine for most cases |
| HTTPd thread pool  in JBoss Web | In the server.xml file under <server>/deploy/  jboss-web-sar. | When making HTTP requests directly to EAP |
| AJP thread pool | In the connector section of server.xml | When making HTTP requests through mod\_jk or mod\_cluster |
| JCA thread pool (also called the  Work Manager thread pool) | Defined in <server>/deploy/  jca-jboss-beans.xml | In conjunction with JMS, as JBoss Messaging  uses JCA inflow as the integration into EAP |
| JBoss Messaging thread pool  (for remote clients) | Defined in <server>/deploy/messaging/  remoting-bisocket-service.xml | Pools the TCP sockets |
| JBoss Messaging thread pool  (in JVM clients) | All the processing will occur on the JCA thread pool (WorkManager) | Note that if you have a message driven bean,  that invokes other beans, such as stateless session beans, those beans will also run on the  JCA thread pool. |
| EJB 3 (same JVM) | Clients in the same JVM will run on whatever thread pool they are already using | For example, a web request comes in through the AJP connector. When it calls an EJB 3 bean,it will continue executing on the AJP connector  thread pool. |
| EJB (remote clients) | <server>/ejb3/deployer/META-INF/jboss-service.xml |  |

**System Thread Pool:**  By default the JBoss shipped with the Neon uses following System thread pool settings:

<!-- A Thread pool service -->

[**-**](about:blank) <mbean code="**org.jboss.util.threadpool.BasicThreadPool**" name="**jboss.system:service=ThreadPool**">

<attribute name="**Name**">**JBoss System Threads**</attribute>

<attribute name="**ThreadGroupName**">**System Threads**</attribute>

- <!-- How long a thread will live without any tasks in MS-->

<attribute name="**KeepAliveTime**">**60000**</attribute>

- <!-- The max number of threads in the pool -->

<attribute name="**MaximumPoolSize**">**10**</attribute>

- <!-- The max number of tasks before the queue is full -->

<attribute name="**MaximumQueueSize**">**1000**</attribute>

- <!-- The behavior of the pool when a task is added and the queue is full.

abort - a RuntimeException is thrown

run - the calling thread executes the task

wait - the calling thread blocks until the queue has room

discard - the task is silently discarded without being run

discardOldest - check to see if a task is about to complete and enque

the new task if possible, else run the task in the calling thread

-->

<attribute name="**BlockingMode**">**run**</attribute>

</mbean>

**HTTPd Thread Pool:** By default the JBoss shipped with the Neon uses following HTTPd thread pool settings:

<!-- A HTTP/1.1 Connector on port 8080 -->

<Connector protocol="**HTTP/1.1**" port="**8080**" address="**${jboss.bind.address}**" connectionTimeout="**20000**" redirectPort="**8443**" />

**JCA Thread Pool:** By default the JBoss shipped with the Neon uses following JCA (Resource Pool) thread pool settings:

<bean name="**WorkManagerThreadPool**" class="**org.jboss.util.threadpool.BasicThreadPool**">

- <!-- Expose via JMX -->

<annotation>**@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.jca:service=WorkManagerThreadPool", exposedInterface=org.jboss.util.threadpool.BasicThreadPoolMBean.class)**</annotation>

- <!-- The name that appears in thread names -->

<property name="**name**">**WorkManager**</property>

- <!-- The maximum amount of work in the queue -->

<property name="**maximumQueueSize**">**1024**</property>

- <!-- The maximum number of active threads -->

<property name="**maximumPoolSize**">**100**</property>

- <!-- How long to keep threads alive after their last work (default one minute)

-->

<property name="**keepAliveTime**">**60000**</property>

</bean>

The above JCA thread pool has been exposed through JMX console via jboss.jca MBean.

**JBoss Messaging Thread Pool (Remote Clients):** By default the JBoss shipped with the Neon uses following Messaging thread pool settings:

<mbean code="**org.jboss.remoting.transport.Connector**" name="**jboss.messaging:service=Connector,transport=bisocket**" display-name="**Bisocket Transport Connector**">

[**-**](about:blank) <attribute name="**Configuration**">

[**-**](about:blank) <config>

[**-**](about:blank) <invoker transport="**bisocket**">

- <!-- There should be no reason to change these parameters - warning!

Changing them may stop JBoss Messaging working correctly -->

<attribute name="**marshaller**" isParam="**true**">**org.jboss.jms.wireformat.JMSWireFormat**</attribute>

<attribute name="**unmarshaller**" isParam="**true**">**org.jboss.jms.wireformat.JMSWireFormat**</attribute>

<attribute name="**dataType**" isParam="**true**">**jms**</attribute>

<attribute name="**socket.check\_connection**" isParam="**true**">**false**</attribute>

<attribute name="**serverBindAddress**">**${jboss.bind.address}**</attribute>

<attribute name="**serverBindPort**">**${jboss.messaging.connector.bisocket.port:4457}**</attribute>

<attribute name="**secondaryBindPort**">**${jboss.messaging.connector.bisocket.secondary.port:4460}**</attribute>

<attribute name="**clientSocketClass**" isParam="**true**">**org.jboss.jms.client.remoting.ClientSocketWrapper**</attribute>

<attribute name="**serverSocketClass**">**org.jboss.jms.server.remoting.ServerSocketWrapper**</attribute>

<attribute name="**numberOfCallRetries**" isParam="**true**">**1**</attribute>

<attribute name="**pingFrequency**" isParam="**true**">**214748364**</attribute>

<attribute name="**pingWindowFactor**" isParam="**true**">**10**</attribute>

<attribute name="**onewayThreadPool**">**org.jboss.jms.server.remoting.DirectThreadPool**</attribute>

- <!-- End immutable parameters -->

<attribute name="**stopLeaseOnFailure**" isParam="**true**">**true**</attribute>

- <!-- Periodicity of client pings. Server window by default is twice this figure

-->

<attribute name="**clientLeasePeriod**" isParam="**true**">**10000**</attribute>

<attribute name="**registerCallbackListener**">**false**</attribute>

<attribute name="**timeout**" isParam="**true**">**0**</attribute>

- <!-- Number of seconds to wait for a connection in the client pool to become free -->

<attribute name="**connectionWait**" isParam="**true**">**10**</attribute>

- <!-- Max Number of connections in client pool. This should be significantly higher than the max number of sessions/consumers you expect -->

<attribute name="**JBM\_clientMaxPoolSize**" isParam="**true**">**200**</attribute>

- <!-- The maximum time to wait before timing out on trying to write a message to socket for delivery -->

<attribute name="**callbackTimeout**">**10000**</attribute>

- <!-- Increasing the ping values, since the defaults are extremely low and causing connection failures -->

<attribute name="**validatorPingPeriod**" isParam="**true**">**60000**</attribute>

<attribute name="**validatorPingTimeout**" isParam="**true**">**30000**</attribute>

</invoker>

[**-**](about:blank) <handlers>

<handler subsystem="**JMS**">**org.jboss.jms.server.remoting.JMSServerInvocationHandler**</handler>

</handlers>

</config>

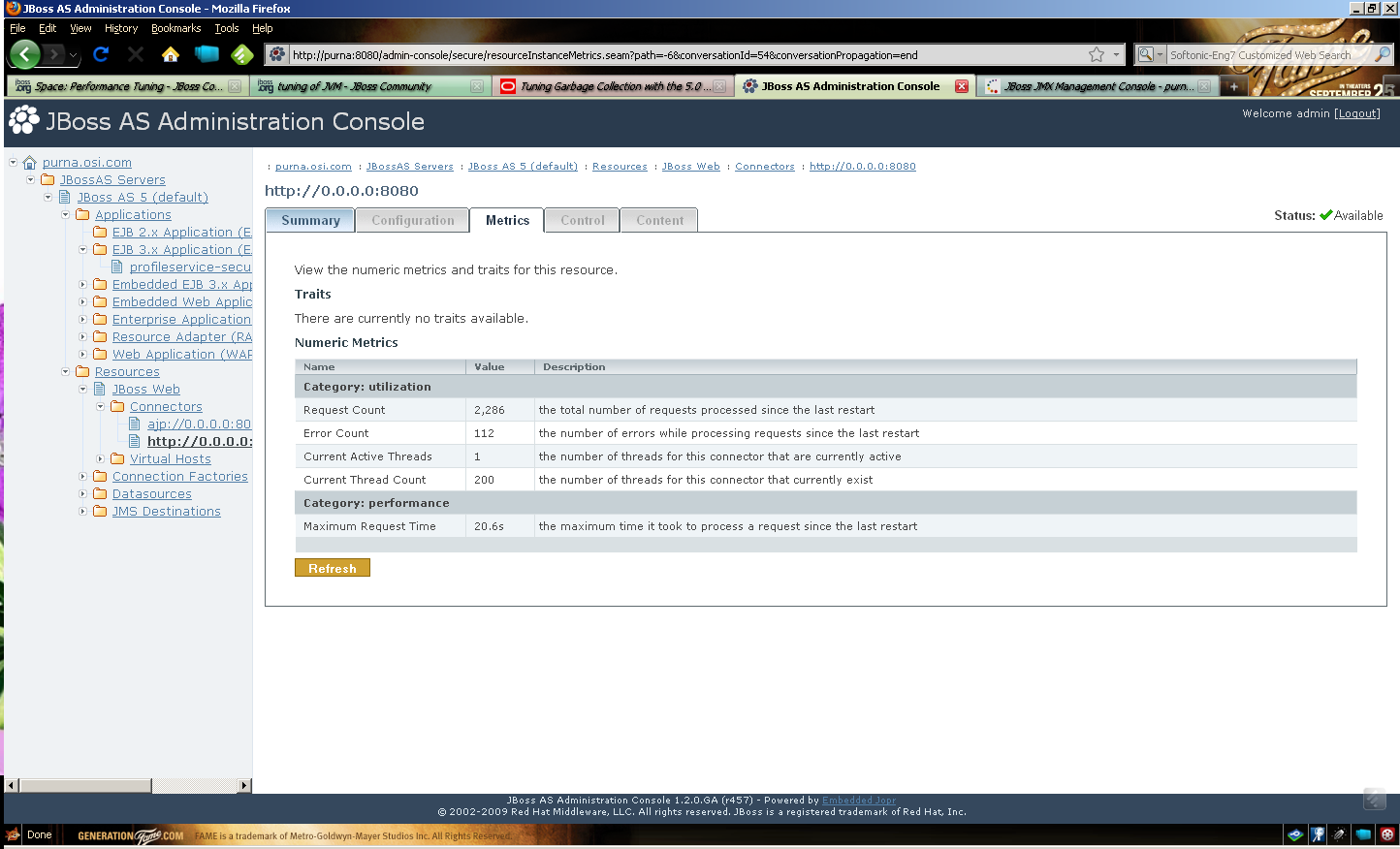
</attribute>

</mbean>

The above Messaging settings have been exposed through JMX xonsole via jboss.remoting MBean

**JBoss Messaging Thread Pool (JVM Clients):** Portal works under HTTP container and do not uses client JVM, all processing happens on JCA thread WorkManager thread pools.

Note: - The thread activity can be monitored using Admin Console:



**3. Object and Component Pool**

Object pools and component pools are essentially the same thing. Their settings represent the number of object instances. For EJB 3, two types of pools are defined in <server>/deploy/ejb3-interceptorsaop.xml. These are the ThreadLocalPool and the StrictMaxPool. By default, Stateless Session and Stateful Session Beans use the ThreadLocalPool, which is backed by an InfinitePool with no maximum size. Therefore, it grows according to volume in your application. This has the distinct advantage of not needing to be tuned.

By default, Message Driven Beans (MDBs) use the StrictMaxPool. This pool actually obeys a maximum, will queue up requests when that maximum has been reached, and will time out anything in the queue if there is not an available reference from the pool. In this case, the system will throw an exception and if the problem occurred in mid-transaction, you will experience a transaction rollback. Given the impact failed transactions can have on your business, you should monitor the **StrictMaxPool** closely via the JMX console.

By default the JBoss shipped with the Neon has following settings in the ejb3-interceptorsaop.xml file:

<annotation expr="**class(\*) AND !class(@org.jboss.ejb3.annotation.Pool)**">**@org.jboss.ejb3.annotation.Pool (value="StrictMaxPool", maxSize=15, timeout=10000)**</annotation>

4. Logging

Following points need to be considered while taking logging into production environments:

* Turn off console logging in production. In JBoss EAPs, console logging is enabled. In production, this is an expensive process with unbuffered I/O
* Turn down logging verbosity. The less you log, the less I/O will occur. By default, the jboss-log4j.xml file contains the logging ‘priority value’ as INFO. The logging level should be changed to ERROR.
* Use asynchronous logging, with asynchronous logging, log messages will go into a queue and control returns to the application as if the logging had been completed. Then a separate thread executes the log operations from the queue. By using queues and topics, Neon seems to be using the asynchronous logging of JBoss.
* A note to developers - Wrap debug log statements with If(debugEnabled()). This simple practice can make a huge difference if your application contains a lot of debug log statements.

**5. Caching**

Cache EJB3 entities to improve performance. To define which entities you want cached, modify the file persistence.xml that you deploy with your EJB 3 application (an example is shown below). You can use the @Cache annotation on the beans you want cached.

You can define the cache as transactional or read-write. With very large data sets, note that caching may not provide noticeable performance benefits. In this case, shrinking the caches can improve performance. Also, if your application is very write-heavy, it may not benefit from caching. Testing various caching and non-caching configurations will help you determine this.

In addition to JBoss Cache, another form of caching that can benefit many applications is prepared statement caching. Prepared statement caching can be set in your data source configuration. By this simple change, it is possible for some applications to experience a significant improvement in throughput.

**6. JVM Tuning**

The JVM provides a runtime execution environment for Java byte code and acts as a layer of abstraction over the operating system. JVMperformance, specifically in the area of garbage collection, can have a significant impact on overall system and application performance.

Before changing JVM parameters (especially Garbage Collection) we should ask **“Whether JVM tuning is necessary for Neon?”** To study how JVM is behaving it is strongly recommend running a load test to verify the duration and frequency of garbage collects. In general, if the load test indicates that an application spends 20% of its time doing garbage collections, that clearly indicates a problem. If an application spends less than 5% of its time doing garbage collections, that clearly indicates that tuning the JVM will not improve performance. If the application spends between 5% and 20% of its time performing garbage collections, then tuning the JVM may improve performance.

To collect GC stats, Set the command line JAVA\_OPTSto identify the time and frequency of garbage collections (for example, "java –verbose:gc –XX:+PrintGCTimeStamps –XX:+PrintGCDetails – loggc=loggc.out" ). Following are the details of these options:

|  |  |
| --- | --- |
| **Option Description** | |
| **-**verbose:gc | Turns on the logging of GCinformation |
| **-**Xloggc=filename | Specifies the name of a log file where the “–verbose:gc” information can be logged (instead of standard output) |
| -XX:+PrintGCTimeStamps | Prints the times at which the GCs happen relative to the start of the application |
| -XX:+PrintGCDetails | Gives detailed information about the GCs, such as size of the young and old generation before and after GCs, size of the total heap, and time that it takes for a GCto happen in the young and old generation. |

Run the load test to determine the frequency and duration of collections.

• Evaluate the frequency of garbage collections and the time spent on each one. A good rule of thumb is to specify minor garbage collections approximately every 30 seconds and full (major) collections more infrequently. View and analyze the **loggc** file to make sure that the heap settles down (rather than grows) after a full collection. You can also use the PrintGCStats tool1 to analyze the total percentage of time the application is spending on garbage collections.

• Adjust the heap size parameters (–Xms, –Xmx, etc.), run the load test, evaluate the time spent in major and minor collections, and adjust the settings until you find those that provide the minimum average overhead from garbage collections.

• If major garbage collections still do not fit within the maximum allowed pause time, then consider other garbage collection algorithms. Normally, the default algorithms work well; however, if you decide to consider other algorithms, then parallel copying collector (–XX:+UseParNewGC) for new generation and the concurrent collector (–XX:+UseConcMarkSweepGC) for old generation may result in performance gains. To study the behavior of the default GC used in the JBoss server and analysis of the garbage collection we can use the VisualGC tool developed by sun to analyse the garbage collection phases and will further find out that if any other garbage collector is required for JBoss.

To use the VisualGC tool, follow the instructions mentioned in the following link: <http://java.sun.com/performance/jvmstat/visualgc.html>

Point the JBoss JVM process to the VisualGC tool and analyze the statistics of the default GC used with the JBoss and find out that if some other GC algorithm is better than what we are shipping with the JBoss. To configure and how to use the VisualGC, the documentation of the same provides all details.

By default, the JBoss shipped with Neon uses following GC parameters:

**JAVA\_OPTS**="$JAVA\_OPTS -Xms512m -Xmx2048m -XX:MaxPermSize=512m -Dorg.jboss.resolver.warning=true -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000 -XX:+HeapDumpOnOutOfMemoryError"

**7. JBoss Clustering**

A cluster is a set of nodes. These nodes generally have a common goal. A node can be a computer or, more simply, a server instance (if it hosts several instances).

In JBoss, nodes in a cluster have two common goals: achieving Fault Tolerance and Load Balancing through replication. These concepts are often mixed.

**Load balancing** is a means to obtain better performance by dispatching incoming requests to different servers. It does not make any assumption on the level of fault tolerance or availability of the system. Thus, a web site could use a farm of servers to render complex output based on basic information stored in a database. If the database is not a bottleneck, load-balancing requests between servers in the farm would largely improve performances.

In several JBoss clustering features, Fail-Over and load-balancing features targets for JNDI, RMI, Entity beans, Stateful Session Beans with in memory state replication, Stateless Session Beans, HTTP Session replication with Tomcat (Probably not useful for Neon as we do not use Tomcat bundled with JBoss), HTTP Session Clustering for Load-balance of incoming invocations (Probably this is also not useful for Neon as we do not deploy any web application in JBoss. We only deploy EJBs in JBoss) and farming. The important ones which can be considered are as follows but these require changes need to be done by the development in their EJB descriptors:

* Entity Beans
* Stateful Session Beans with in memory state replication
* Stateless Session Beans

**Entity Beans**

To cluster an entity bean you need to modify its jboss.xml descriptor to contain a <clustered> tag. For e.g.

<jboss> <enterprise-beans> <entity>

<ejb-name>nextgen.EnterpriseEntity</ejb-name>

<jndi-name>nextgen.EnterpriseEntity<jndi-name>

**<clustered>True</clustered>**

**<cluster-config>**

**<partition-name>DefaultPartition</partition-name>**

**<home-load-balance-policy>org.jboss.ha.framework.interfaces.RoundRobin**

**</home-load-balance-policy>**

**<bean-load-balance-policy>org.jboss.ha.framework.interfaces.FirstAvailable**

**</bean-load-balance-policy>**

**</cluster-config>**

**</entity></enterprise-beans> </jboss>**

Clustered Entity Beans do not currently have a distributed locking mechanism or a distributed cache. They can only be synchronized by using row-level locking at the database level or by setting the Transaction isolation level of the JDBC driver to be TRANSACTION\_SERIALIZABLE.

**Stateful Session Beans**

Clustering stateful session beans has much more implications than clustering stateless beans: we need to manage state!

In the current implementation we do not use any database or other equivalent mechanism to replicate and share the state of beans. Instead, we use in-memory replication between nodes. The state of all SFSBs are replicated and synchronized across the cluster each time the state of a bean changes.

To manage the Stateful Session Bean state, a cluster-wide distributed service is needed. This service is a Jboss Mbean called HASessionState:

<mbean

code="org.jboss.ha.hasessionstate.server.HASessionStateService" name="jboss:service=HASessionState">

</mbean>

Then each stateful session bean needs to modify its scriptor to contain a *<clustered>* tag.

<jboss> <session> <ejb-name>nextgen.StatefulSession</ejb-name>

<jndi-name>nextgen.StatefulSession</jndi-name>

**<clustered>True</clustered>**

**</cluster-config>**

**<partition-name>DefaultPartition</partition-name>**

**<home-load-balance-policy>org.jboss.ha.framkework.interfaces.RoundRobin**

**</home-load-balance-policy>**

**<bean-load-balance-policy>**

**org.jboss.ha.framkework.interfaces.FirstAvailable**

**</bean-load-balance-policy>**

**<session-state-manager-jndi-name>**

**/HASessionState/Default**

**</session-state-manager-jndi-name>**

**</cluster-config>**

</session>

</enterprise-beans>

</jboss>

Actions on the clustered SFSB’s home interface are by default load-balanced, round-robin. Once the bean’s remote stub is available to the client, calls will not be load-balanced round-robin anymore and will stay “sticky” to the first node in the list.

As the replication process is a costly operation, you can optimize this behavior by implementing in your bean class a method with the following signature:

public boolean isModified ();

Before replicating your bean, the container will detect if your bean implements this method and possibly call it. If the bean has not bean modified (or not enough to require replication, depending on your own preferences), the replication will not occur.

**Stateless Session Beans**

Clustering stateless session beans is most probably the easiest case: as no state is involved, calls can be, à priori, load-balanced on any participating node (i.e. any node that has this specific bean deployed) of the cluster.

To make a bean clustered, you need to modify its *jboss.xml* descriptor to contain a *<clustered>* tag.

<jboss>

<enterprise-beans>

<session>

<ejb-name>nextgen.StatelessSession</ejb-name>

<jndi-name>nextgen.StatelessSession</jndi-name>

**<clustered>True</clustered>**

**<cluster-config>**

**<partition-name>DefaultPartition</partition-name>**

**<home-load-balance-policy>**

**org.jboss.ha.framework.interfaces.RoundRobin**

**</home-load-balance-policy>**

**<bean-load-balance-policy>**

**org.jboss.ha.framework.interfaces.RoundRobin**

**</bean-load-balance-policy>**

**</cluster-config>**

</session>

</enterprise-beans>

</jboss>

The *<partition-name>* tag is used to determine in which cluster the bean will participate. It uses by default, the default partition.

The *<home-load-balance-policy>* indicates the class to be used by the home proxy to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a round-robin fashion. You can also implement your own load-balance policy class or use the class org.jboss.ha.framework.interfaces.FirstAvailable that persist to use the first node available that it meets until it fails.

The <bean-load-balance-policy> indicates the class to be used by the remote proxy to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a round-robin fashion. Comments made for the *<home-load-balance-policy>* tag also apply.

**8. Latest JDBC drivers**

JBoss 5.1.0 is using Oracle 10g driver (10.2.0.3.0) to connect to Oracle 11g. The ojdbc14.jar (JBOSS\_HOME/common/lib) is for jdk 1.3 & jdk1.4. We should move to Oracle 11g JDBC driver for jdk 1.6 i.e. ojdbc16.jar

**Tomcat (Portal Server) performance tuning areas**

The tomcat (Tomcat 6) which is shipped with the Neon is been customized to lay out j2-admin and jetspeed applications as it is. These applications are not deployed when user starts tomcat (Portal Server). The Portal Server hosts SMF application and contacts connected JBoss to access EJBs to provide services to portal client.

Following are the performance tuning areas which needs to be considered while tuning Portal Server:

1. Multiple Tomcat Instances (Load balancing)
2. Increasing request threads
3. Decreasing connection timeout for highly concurrent environment
4. Turning off KeepAlive
5. Tune logging parameters
6. Tuning content cache
7. JVM Tuning (If Required)
8. **Multiple Tomcat Instances (Load balancing)**

Tomcat being used in Neon starts processing its requests through Coyote server running on default port 8010. The tomcat configuration has been customized as per the JBoss architecture and does not allow to add separate modules as such which user can do easily by providing a httpd.conf file.

To configure load balancing on the portal server side we either have to use **mod\_rewrite** with **mod\_proxy** or **mod\_jk.** Both these options require httpd daemon to be exposed as it is exposed in the fresh build of tomcat.

To configure mod\_jk, A shared library needs to be downloaded which contains modules supported for mod\_jk and to route HTTP requests above portal server as AJP requests. For this to happen, AJP protocol connector needs to be uncommented in the neon-portal/conf/server.xml and neon-portal/conf/server-default.xml files. Allowing AJP in Portal server will make TCP connections with the servlet container (JBoss) and these connections will be persistent.

In brief, the Portal Server needs some changes in order to implement mod\_jk.

1. **Increasing request threads**

Increasing thread count to process more requests will help to increase tomcat performance but the underlying machine resources i.e. CPU, Memory also needs to be considered. A parameter provided in the **neon-portal/conf/server-default.xml** needs to be changed to do this:

<Service name="Catalina">

<!--The connectors can use a shared executor, you can define one or more named thread pools-->

<!--

<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"

**maxThreads**="150" minSpareThreads="4"/>

By default, the maxThreads value is 150 and a good starting value is 400

1. **Decreasing connection timeout for highly concurrent environment**

Following are the settings to decrease the value of connectionTimeOut parameter.

<Connector port="8080" protocol="HTTP/1.1"

**connectionTimeout**="20000"

redirectPort="8443"

emptySessionPath="true" />

<!-- A "Connector" using the shared thread pool-->

<!--

<Connector executor="tomcatThreadPool"

port="8080" protocol="HTTP/1.1"

**connectionTimeout**="20000"

redirectPort="8443" />

Default value of 20,000 (20 secs) is too high for a web server.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*