Edition

1

Date: 2004-05-21, 4:04:58 PM

ADRIAN BROCK

The JBoss Group

JBoss Tuning

ADRIAN BROCK AND THE JBOSS GROUP

JBoss Tuning

2520 Sharondale Dr.
Atlanta, GA 30305 USA
sales@jbossgroup.com

Table of Contents

Table of Listings

Table of Tables

Preface

Forward

TODO

About the Authors

Adrian Brock was born in Doncaster, England in 1969. He has a degree in Mathematical Physics from Manchester University. He is currently the Director of Support for JBoss Group and the lead for JBossMQ and JCA. He also helps to fix and test most parts of JBoss.

The JBoss Group, LLC, headed by Marc Fleury, is composed of over 1000 developers worldwide who are working to deliver a full range of J2EE tools, making JBoss the premier Enterprise Java application server for the Java 2 Enterprise Edition platform.

JBoss is an open source, standards-compliant, J2EE application server implemented in 100% Pure Java. The JBoss/Server and complement of products are delivered under a public license. With 150,000 downloads per month, JBoss is the most downloaded J2EE based server in the industry.

Acknowledgments

Thanks to the rest of JBoss Group who provided much of the information in this document. The CMP optimized loading section was written by Dain Sundstrom.

Chapter

0

0. Introduction

What this Book Covers

This book covers the tuning of the JBoss Application Server and related Operating System tuning. It assumes knowledge of JBoss and the Operating Systems discussed.

It is not the intention of this document to provide complete configuration details for JBoss. Details can be found in the Admin and Development documentation. Instead, it isolates those configurations that affect tuning.

This document is not an introduction to tuning. There is discussion on how the configurations affect the performance of JBoss that provide some details on how the tuning works.

Finally, there is no discussion on performance tools. Performance tools are very important. No amount of theory can replace actually measuring how the tuning parameters affect reality.

Organization

The document starts with generic ideas before going into specific areas in more detail.

Chapter 1, Concepts

Techniques and concepts used in tuning and application design

Chapter 2, Operating systems

The impact of the operating system

Chapter 3, Virtual machines

Virtual machines is the next abstraction above the operating system

Chapter 4, EJB Containers

Configuration of EJBs, pooling, caching and locking

Chapter 5, CMP Optimized loading

Optimizing queries and bean data loading

Chapter 6, Web Containers

Thread Pooling and other optimizations

Chapter 7, The Front End

Remote invocations

Chapter 8, The Back End

Databases and connection pooling

Chapter 9, Miscellaneous

JNDI, Transaction Manager and JMS

Appendix A, About The JBoss Group

This appendix contains information about The JBoss Group.

Chapter

1

1. Concepts

Before we start, it should be made clear that although performance is important, do not sacrifice correctness or stability to achieve it.

The aim of tuning is to make a software deployment perform in the most efficient manner. The meaning of efficiency depends upon what you are trying to achieve. In many circumstances, the aims are contradictory, you have to choose what you are trying to achieve. Some subsystems can have different and competing aims.

To perform tuning correctly you have to understand where the bottlenecks are located. The often quoted "Don't optimize too early" maxim is normally true. Equally, a bad early design will also cause you problems. The design should consider how to avoid potential obvious problems, but only measurement will show where the true bottlenecks are located.

Aims

Guaranteed Response Time

J2EE application servers are not really designed as real time systems and neither are most operating systems. Guaranteed small response times are not possible.

Reducing Response Time

One option for tuning is to try to reduce the average response time for requests. This usually involves avoiding intensive processing in interactive requests. Intensive processes are delegated to background subsystems. In practice, the overhead of the asynchronicity framework will use more CPU than if the request was allowed to complete in line, reducing the throughput. Similar considerations apply to lots of small requests versus one big request.

Increasing Throughput

Another option is to process as many transactions as possible. This involves using resources in the most efficient manner without worrying too much about long response times. Throughput

and response time are not mutually exclusive, especially when long running processes hold locks that reduce concurrency.

Bottlenecks

CPU

A computer has only so much CPU. This is a scalability issue. Once you reach 100% CPU utilization no more processing can be done. Sometimes it is worthwhile to do extra processing that will avoid other problems.

Memory

Memory is also a limited resource. All modern operating systems provide virtual memory paging to extend this limited resource. In many cases, the paging is a source of bad performance. Memory is a relatively cheap resource. Using more memory to avoid other problems is often a good strategy, but this can only go so far with limited address spaces and the scalability of the garbage collector.

Threads

Threads form a major part of an application server that processes multiple requests concurrently. They are a limited resource. Each thread has to be handled by the operating system's scheduler. Some modern operating systems have very efficient scheduling routines, but they can still be starved of the limited CPU. Additionally, each thread has a stack that holds methods parameters, local variables and a work area for operations that cannot are not handled directly by the CPU. Each stack consumes memory.

Communication/Serialization

Communication with other machines involves latency, which affects response times. This becomes more significant when there are many small requests versus a few big requests. Other latencies occur when the communication performs buffering. A request or response might be completed but it is sitting inside an operating system buffer waiting for the buffer to fill.

Less obviously, communication is done in bytes whether it is a raw byte stream or some other format like xml. The conversion of java objects to a byte format and back again consumes a lot of CPU. This is called Serialization.

Locking

In any concurrent system, access to shared resources has to be controlled. This leads to some threads waiting for a resource to become available while another uses it. A shared resource can become a global point of contention when everybody is waiting for it. This turns your multi-threaded application server into a single threaded machine

Techniques

Pooling

A pool is a group of objects that are all equivalent. The key elements for using a pool of objects are:

- One object can be used interchangeably with another.
- The object can be expensive to construct, compared with its use.
- Each object has a mechanism to remove any transient state so that it can be placed back in the pool after use.

The main advantages of a pool are:

- The repeated expensive construction of the object is avoided
- Resource usage can be limited using a strict a pool. When all the objects in the pool are in use, the request waits

The main disadvantages of a pool are:

- They unnecessarily allocate resources when the pool is hardly used. This can be negated by having a reaper process for idle objects.
- They can be an unexpected source of error when the contract of equivalence is violated.
- A pool with a strict maximum size that is too small becomes a point of contention.

Pooling a common pitfall

Pools cannot be considered in isolation. It is no good having lots of threads waiting for http requests on the front end when there are too few database connections on the backend to process those requests.

Caching

Caching involves storing state or the result of an operation against a unique identifier. The aim of caching is to avoid repeating an expensive operation. Instead, the cache is used to lookup the result. The key elements for using a cache are:

- The lookup replaces an expensive operation.

- The operation can be uniquely identified, e.g. a database row primary key.

The advantages of a cache are:

- An expensive operation is replaced with a cheap operation.
- One thread can do the work of many, i.e. the first performs an operation while the others wait for the result.

The disadvantages of a cache are:

- It uses more resources in terms of memory. Mechanisms must be put in place to avoid the cache blowing the memory.
- The cache can become dirty. I.e., it is no longer a true representation of the state or the operation.
- Mechanisms used to avoid a dirty cache and ensure cache consistency can become complicated or lead to other bottleneck problems.

General Cache Considerations

Whether a cache can be used and what type of cache is normally driven by the type of data. For example, take some common categories of data.

- Bank account data – the program cannot deal with inconsistent data for even the shortest amount of time. When an amount is transferred from one account to another, the program must be dealing with the real figures and must prevent changes made by other processes until it has finished. In this case, a cache can only be used if it is aware of transactions. The cache also needs to co-operate in the locking.
- Read mostly – data that changes infrequently and the timing of when the change is seen is not strictly important. An example is a customer addresses. A cache can be used in this case provided there is a cache invalidation when somebody changes the address. This is because there is no circumstance when the change of address would be tied transactionally with the orders on which it is used.
- No updates – data is only ever inserted. Since data does not change, the cache is always valid.
- Idempotent – An operation always returns the same result. E.g., calculate the profit from a closed accounting period. The cache is always valid.

Clustering

There are three different elements to clustering:

- Load balancing – distributing requests across a cluster of machines
- Failover – when one machine fails, the request is rerouted to another machine
- Replication – to support failover for a stateful service, the state must be replicated to other machines.

When implementing clustering it is important to consider what you are trying to achieve.

- Scaling – You want to process more requests by making more machines. This involves load balancing.
- Reliability – You want to avoid single point of failures. This is done using failover and replication.

While not necessarily mutually exclusive, it is difficult to scale when state has to be replicated to more and more machines.

Stateless designs allow scaling.

Stateful designs require reliability.

The state can be replicated using a shared resource, e.g. entity beans sharing the same database. This negates the need for replication allowing failover and scaling. You will notice the single point of failure is the database. Reliability requires a clustered database solution.

Chapter 2

2. Operating Systems

The operating system is a key component. Ultimately, a poorly configured operation system will outweigh any improvements you make in your application.

Some points that may appear obvious but are often overlooked:

- If there are other processes running on the same server, you are limiting the resources available to the application server.
- Do not let the system administrator play games on the production server.
- Do not run an FTP server that is going to use up your network bandwidth and thrash the disk.

CPU

Application servers do not use a lot of CPU compared with other programs. Most CPU is used serializing data over the network or with the persistent store. There are two caveats to this statement:

- The hosted applications may use a lot of CPU, e.g. XSL transforms
- A properly tuned application server that avoids contention should be able to scale up to 100% CPU utilization because of the number of requests it is processing.

Memory

Other processes using memory will cause the application server's memory to be paged out to disk. This causes a problem for the garbage collector, which we will discuss later.

Some OS configurations have a direct affect on memory.

Windows

Changing the memory model to use /3GB can result in less page tables being available. See the following link: http://support.microsoft.com/default.aspx?scid=kb;en-us;313707

Linux – Stack Sizes

The default stack size on most Linux distributions is 8MB. This makes sense for many UNIX programs that allocate large data structures on the stack and only use a single thread. Java allocates most memory in the heap. The stack is largely made up of primitives like int and long or object references into the heap, they are only 4 or 8 bytes in size. These are method parameters or local variables. In practice, a Java thread stack will never approach 8MB. This becomes more important when you consider 10 threads allocate 80MB for the stacks. It is easy to see that it becomes difficult to create many threads with so much memory being allocated. The stack allocation can be reduced, e.g. ulimit –s 2048 will make the stacks 2MB in size allowing four times the number of threads in the same memory. NOTE: There is an ulimit command in run.sh

Linux – Over-commit

The Linux kernel uses a memory model called over-commit. The idea is that most programs allocate more memory than they use. The kernel remembers that you asked for the memory, but does not allocate the memory until you actually use it. This is ok unless all allocations are actually used. If this happens, there will not be enough virtual memory available. The kernel resolves the problem by killing a process. In the 2.6 kernel, sysctl can be used to configure this. In earlier kernels, a patch is required.

Network

A network is not a concurrent object. Only one process can use it at once. TCP/IP has a threading/scheduling notion in its protocol stack, but this is an illusion like threads sharing a single CPU. Other processes using the network card will restrict the communication with the application server.

Other things to consider are how the application controls the sockets.

Windows – Dynamic backlog

A common problem with Windows is the backlog processing. By default, it has a limited backlog that can quickly produce "Connection Refused" under load. Information on dynamic backlogging can be found here: http://support.microsoft.com/default.aspx?scid=kb;en-us;142641

Windows – Anonymous ports

Windows has problems with a limited number of anonymous ports. See the link http://support.microsoft/com/default.aspx?scid=kb;en-us;319502

Linux – File Descriptors

Most distributions have a maximum file descriptors per process set to 1024. This is important because everything in UNIX is a file, so this affects the number of sockets that can be opened.

The limit can be changed using ulimit –n, but you cannot increase it without first adding some configuration to /usr/security/limits.conf. Note: There is a ulimit command in run.sh

Additionally, there is a global limit for total open files in the system. To view, use the command cat /proc/sys/fs/file-max to change cat new-value > /proc/sys/fs/file-max

Firewalls

Check for unnecessary buffering by the firewall or proxy, this will introduce latency.

Make sure any stateful firewall does not timeout connections due to inactivity. This is important for things like modjk that keeps a connection open between Apache and JBoss.

Disks

Separating the Load

Disks have a limited number of heads that can read or write data. Putting all the files on the same disk makes it easy for backup, but it will also reduce the concurrency. Separate things like the web server access log and transaction logs onto different disks. NOTE: Many disks are partitioned into multiple file systems. They are still physically the same disk.

CRON Jobs

Linux distributions are typically configured with CRON jobs. These do admin processes like clearing the temporary directory or analyzing the file system for consistency, search indexing or security problems.

The problem with clearing the temporary directory is that java might have put files there. Define the system parameter –Djava.io.dir=/some/where/else to avoid this problem.

The analysis of the file system will typically thrash the disk, making it hard for the application server to access it. Linux uses a buffer cache to load part of the disk into memory. The file system analysis will replace any cached files used by the application server with files used in the analysis.

Chapter 3

3. Virtual Machines

Tuning

This document will not go into the tuning of individual virtual machines. Each virtual machine is different. Instead, it will identify what you need to configure.

Memory

Heap sizes

The main thing to configure is the heap. There are typically settings for minimum and maximum heap sizes. The default for maximum heap on Sun virtual machines is quite small (64MB). It is normally a good idea to set the minimum and maximum to the same value. This guarantees when the virtual machine starts that enough memory is available. This avoids surprises later when it tries to allocate the full memory.

The heap itself is normally divided up into areas. The commonly used generational garbage collectors have multiple areas in the heap used for objects of different ages.

There are other areas of memory that produce OutOfMemoryExceptions when they are too small. One common reason for OutOfMemoryException on Sun virtual machine is insufficient permanent generation size. The virtual machine uses the permanent generation to store reflective data, such as the `Class` and `Method` instances. You can allocate a larger permanent generation with –XX:MaxPermSize in the Sun VM.

An object is first created in the younger generations where it is efficient to deallocate the object. It is promoted into older generations the longer it survives.

Making more memory available to the younger generations will reduce the number of garbage collections. However, the older generations will fill up more quickly, making full garbage collections more common. The opposite is true. A smaller younger generation leads to more collections but less full garbage collections.

Full garbage collections occur when the heap has run out of memory. These take a long time because all references are analysed. Also, the full garbage collection algorithms often attempt to

compact the memory while collecting the garbage to fight memory fragmentation. This further increases the time it takes to finish a full garbage collection.

Some applications instruct the VM when a garbage collector should be run (System.gc() call). This instruction, if followed by the VM, will lead to a full garbage collection. Executing the full collection too frequently will essentially render the young generations useless and affect negatively the application server's throughput.

Sun RMI implementation is one such application that frequently instructs the JVM to do a full garbage collection (with default settings, once every 60 seconds). It is recommended that the explicit GC usage is disabled whenever possible (use –XX:+DisableExplicitGC on the Sun VM).

Types of Garbage Collector

There are two main types of garbage collector.

- Throughput garbage collectors do all the work in one pass. A single collection takes longer but it is the most efficient.
- Incremental garbage collectors do partial collections. This reduces the amount a collection takes. This reduces the response time at the expense of throughput.

More recently, concurrent and background garbage collectors have become available. These divide the work amongst many threads and even do some work while the application is running. Older implementations were mainly single threaded and "stop the world", i.e. the application was paused during the garbage collection. The advantage of these collectors only becomes apparent on multi-processor machines. Single and dual processor machines can run slower with these configurations.

One of the recent Sun garbage collectors (the concurrent throughput collector) is self-tuning. That is, it records information about previous collections to try to optimize the collection and heap allocation strategy.

Garbage Collectors and paging

Running a garbage collector on a system that is swapping virtual memory to disk and vice-versa causes performance problems. When the system wants to do a full garbage collection it will look at the whole heap, some of which is on disk. This will lead to starvation where the application is waiting for garbage collector that in turn is waiting for the disk.

RMI

The default invoker within JBoss uses RMI. This means that the amount of tuning depends upon what is provided by the virtual machine. Where available, configure the rmi thread pool.

The backlog can be configured within JBoss by configuring your own socket factory. It is also possible to use this configuration to add behaviour like compressed streams. Compressing streams is a trade-off between the CPU required to do the compression/decompression against time saved sending the data over the network.

Chapter

4

4. EJB Containers

This section discusses the tuning of EJB Containers. Remote invocation of EJBs is discussed in a later section. The examples configurations are based on those in conf/standardjboss.xml. These can be overridden in META-INF/jboss.xml. The full list of possible configurations can be found in docs/dtd/jboss_3_2.dtd

Stateless session beans

SLSBs are relatively simple. They are little more than a pool of bean instances that can efficiently implement a process. Typically, they delegate real work to some other object.

Pooling

The most important configuration is the pool:

```
      <container-configuration>

         <container-name>Standard Stateless SessionBean</container-name>

         <call-logging>false</call-logging>

         <invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>

         <container-interceptors>

            <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>

            <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>

            <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>

            <!-- CMT -->

            <interceptor transaction="Container">org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

            <interceptor transaction="Container"
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>

            <interceptor
transaction="Container">org.jboss.ejb.plugins.StatelessSessionInstanceInterceptor</interceptor>

            <!-- BMT -->

            <interceptor
transaction="Bean">org.jboss.ejb.plugins.StatelessSessionInstanceInterceptor</interceptor>

            <interceptor transaction="Bean">org.jboss.ejb.plugins.TxInterceptorBMT</interceptor>
```

```
            <interceptor transaction="Bean"
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>


<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>

         </container-interceptors>

         <instance-pool>org.jboss.ejb.plugins.StatelessSessionInstancePool</instance-pool>

         <instance-cache></instance-cache>

         <persistence-manager></persistence-manager>

         <container-pool-conf>

            <MaximumSize>100</MaximumSize>

            <strictMaximumSize/>

            <strictTimeout>30000</strictTimeout>

         </container-pool-conf>

      </container-configuration>
```

Listing 4-1, Stateless session bean pool configuration

- <MaximumSize> represents the number of objects in the pool. Without <strictMaximumSize/> configured, this represents the number of objects that will be returned to the pool. The number of active objects can exceed the maximum if there are more requests, only the <MaximumSize> objects are returned to the pool.
- <strictMaximumSize> enforces that only <MaximumSize> objects will be active. Any subsequent requests will wait for an object to be returned to the pool.
- <strictTimeout> is the time to wait for an object to be made available when using <strictMaximumSize>

Caching

There is no caching for SLSBs. All beans are equivalent. There is no unique identity.

Transactions

It is a common mistake to use the default "Required" attribute for transactions. In many cases, an invocation on a session method does not require a transaction. An classic example is session.create()

Proxies

All remote and local interfaces for a stateless session bean are equivalent. They just provide access to the pool. The proxy is also stateless. This means it can be used concurrently by many threads. This makes the proxy ideal for caching rather than repeatedly invoking home.create(). This

behaviour is not guaranteed by spec, caching the home interface is part of the spec. You can take advantage of this behaviour without changing your code by changing the proxy configuration. The StatelessSessionHomeInterceptor caches the result of session.create() after the first invocation. This assumes you have implemented home interface caching.

```
<invoker-proxy-binding>
   <name>stateless-rmi-invoker</name>
   <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
   <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
   <proxy-factory-config>
      <client-interceptors>
         <home>
            <interceptor>org.jboss.proxy.ejb.StatelessSessionHomeInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
         </home>
         <bean>
            <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
         </bean>
      </client-interceptors>
   </proxy-factory-config>
</invoker-proxy-binding>
```

Listing 4-2, Stateless session bean pool configuration

Non Stateless, stateless session beans

The stateless behaviour describes the client's view of the bean. The client should not be aware of any state inside the bean. E.g., implementing an invocation count in the bean and letting the client retrieve it breaks the contract.

This bean can still have state that is not visible to the client. For example, it can implement a cache or it can cache the results jndi lookups for other beans or resources it uses. It is important that all objects are the same so it is a good idea to make this state static, but you need to ensure they can be used concurrently without error or introducing points of contention.

Clustering

SLSBs scale very well because they have no state. It does not matter which machine in the cluster you contact. All the beans are equivalent. This makes them ideal for load balancing.

Message Driven Beans

MDBs are very similar to stateless beans. The main difference is that they are not invoked by clients. Instead, the messaging system delivers messages to the MDB when they are available.

Pooling and caching

The pooling and caching considerations are the same as stateless session beans

Delivery

The delivery also involves a pool. You can think of it as a pool of java.jms.XASessions.

```
<invoker-proxy-binding>
   <name>message-driven-bean</name>
   <invoker-mbean>default</invoker-mbean>
   <proxy-factory>org.jboss.ejb.plugins.jms.JMSContainerInvoker</proxy-factory>
   <proxy-factory-config>
      <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>
      <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
      <MaximumSize>15</MaximumSize>
      <MaxMessages>1</MaxMessages>
      <MDBConfig>
         <ReconnectIntervalSec>10</ReconnectIntervalSec>
         <DLQConfig>
            <DestinationQueue>queue/DLQ</DestinationQueue>
            <MaxTimesRedelivered>10</MaxTimesRedelivered>
            <TimeToLive>0</TimeToLive>
         </DLQConfig>
      </MDBConfig>
   </proxy-factory-config>
</invoker-proxy-binding>
```

Listing 4-3, Message driven bean delivery configuration

- MaximumSize – this is the maximum number of sessions, it is a strict pool.
- MaxMessages – this is used to avoid context switching, the same session is reused to deliver multiple messages on the same thread. This can introduce latency because MaxMessages messages must be delivered before any processing is performed. The messages are delivered in separate transactions.

Stateful Session Beans

SFSBs can be big problems in an application server. The management of the state introduces many complications.

Pooling

The pooling is the same as stateless session beans. However, strict pooling makes less sense unless you want to restrict the number of "active" instances of a stateful session bean.

Caching

The cache is used to store the state of SFSBs between invocations. To avoid the state overflowing memory, beans within the cache can be passivated to disk. If they continue to be unused (because the client has lost interest), the passivated bean is removed from disk to avoid it becoming full.

```
      <container-configuration>

         <container-name>Standard Stateful SessionBean</container-name>

         <call-logging>false</call-logging>

         <invoker-proxy-binding-name>stateful-rmi-invoker</invoker-proxy-binding-name>

         <container-interceptors>

            <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>

            <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>

            <!-- CMT -->

            <interceptor transaction="Container">org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>

            <interceptor transaction="Container"
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>

            <interceptor
transaction="Container">org.jboss.ejb.plugins.StatefulSessionInstanceInterceptor</interceptor>

            <!-- BMT -->

            <interceptor
transaction="Bean">org.jboss.ejb.plugins.StatefulSessionInstanceInterceptor</interceptor>

            <interceptor transaction="Bean">org.jboss.ejb.plugins.TxInterceptorBMT</interceptor>
```

```
            <interceptor transaction="Bean"
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>


<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>

            <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>

         </container-interceptors>

         <instance-cache>org.jboss.ejb.plugins.StatefulSessionInstanceCache</instance-cache>

         <persistence-
manager>org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager</persistence-manager>

         <container-cache-conf>

            <cache-policy>org.jboss.ejb.plugins.LRUStatefulContextCachePolicy</cache-policy>

            <cache-policy-conf>

               <min-capacity>50</min-capacity>

               <max-capacity>1000000</max-capacity>

               <remover-period>1800</remover-period>

               <max-bean-life>1800</max-bean-life>

               <overager-period>300</overager-period>

               <max-bean-age>600</max-bean-age>

               <resizer-period>400</resizer-period>

               <max-cache-miss-period>60</max-cache-miss-period>

               <min-cache-miss-period>1</min-cache-miss-period>

               <cache-load-factor>0.75</cache-load-factor>

            </cache-policy-conf>

         </container-cache-conf>
```

Listing 4-4, Stateful session bean cache configuration

- <min-capacity>, <max-capacity>, <max-cache-miss-period>, <min-cache-period> and cache-load-factor are used to dynamically size the cache. The idea is that if there are many cache misses more beans will be kept in memory. Cache hits will lead to beans being removed from cache. This resizing is run periodically based on the <resizer-period>. The actual algorithm is quite complicated.

- <max-bean-age> and <overager-period> control when unused beans are written to disk. The default configuration is to check every 5 minutes for beans that have not been used for 10 minutes.

- <max-bean-life> and <remove-period> control when unused beans are removed from disk. The default configuration is to check every 30 minutes for beans that have not been used for 30 minutes.

Contention

SFSBs cannot be accessed concurrently on multiple threads.

BMT SFSBs can hold a user transaction open over multiple invocations, but once a transaction is active for an instance, another transaction cannot access the bean, even when the original transaction is not currently using the instance.

Clustering

For reliability, the state of the SFSB must be replicated across the cluster. This involves passivating the instance to get a serialized copy of the bean state.

If there is other implementation state for the bean, this has to be reconstructed on the next invocation.

SFSBs with large states will use a lot of network bandwidth during the replication. Where possible try to differentiate state that can be reconstructed using other methods from state that is not stored elsewhere. This will minimize the amount of replicated state.

In some cases, it does not matter that the state is lost on failover. E.g., the SFSB is just being used as cache object for the client. All state can be reproduced after a crash.

Entity Beans

Entity Beans are used to represent persistent state. They form natural caches when configured that way.

Pooling

The pooling configuration is the same as Stateful session beans

Caching

The entity bean cache is similar to the Stateful session bean cache. The difference is that state is already on disk. Passivation just involves removing the instance from the cache. It can be reloaded from the persistent store.

```
<container-configurations>
```

```
      <container-configuration>
         <container-name>Standard CMP 2.x EntityBean</container-name>
         <call-logging>false</call-logging>
         <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-name>
         <sync-on-commit-only>false</sync-on-commit-only>
         <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
         <container-interceptors>
            <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
            <interceptor metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>
<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
         </container-interceptors>
         <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
         <instance-cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instance-cache>
         <persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-manager>
         <locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-policy>
         <container-cache-conf>
            <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
            <cache-policy-conf>
               <min-capacity>50</min-capacity>
               <max-capacity>1000000</max-capacity>
               <overager-period>300</overager-period>
               <max-bean-age>600</max-bean-age>
               <resizer-period>400</resizer-period>
               <max-cache-miss-period>60</max-cache-miss-period>
```

```
            <min-cache-miss-period>1</min-cache-miss-period>
            <cache-load-factor>0.75</cache-load-factor>
         </cache-policy-conf>
      </container-cache-conf>
      <container-pool-conf>
         <MaximumSize>100</MaximumSize>
      </container-pool-conf>
      <commit-option>B</commit-option>
   </container-configuration>
```

Listing 4-5, Entity bean cache configuration

- <min-capacity>, <max-capacity>, <max-cache-miss-period>, <min-cache-period> and cache-load-factor are used to dynamically size the cache. The idea is that if there are many cache misses more beans will be kept in memory. Cache hits will lead to beans being removed from cache. This resizing is run periodically based on the <resizer-period>. The actual algorithm is quite complicated.

- <max-bean-age> and <overager-period> control when unused beans are written to disk. The default configuration is to check every 400 seconds for beans that have not been used for 10 minutes.

- <max-bean-life> and <remove-period> control when unused beans are removed from disk. The default configuration is to check every 30 minutes for beans that have not been used for 30 minutes.

That covers the basic cache configuration. Entity bean caching has a lot more complexity.

First, there are commit options that controls what happens to the bean at the end of transaction. Second there are alternate configurations that do not use the above the cache configuration. Finally, there are a number of different cache invalidation configurations. The cache configuration is linked to the locking.

It is important to remember that the type of data drives which configuration should be used.

Commit Options

- Option A – always cache the bean instance and its data. This can be used when JBoss knows about all changes to the bean's state.

- Option B – keep hold of the bean instance in the cache, but mark the data as invalid at the end of the transaction. This should be used when JBoss does not know about changes to

the bean's state but the bean instance is expensive to construct. You should note that there is a single shared instance of the bean in the cache. Access to the bean is queued on a lock to ensure consistency.

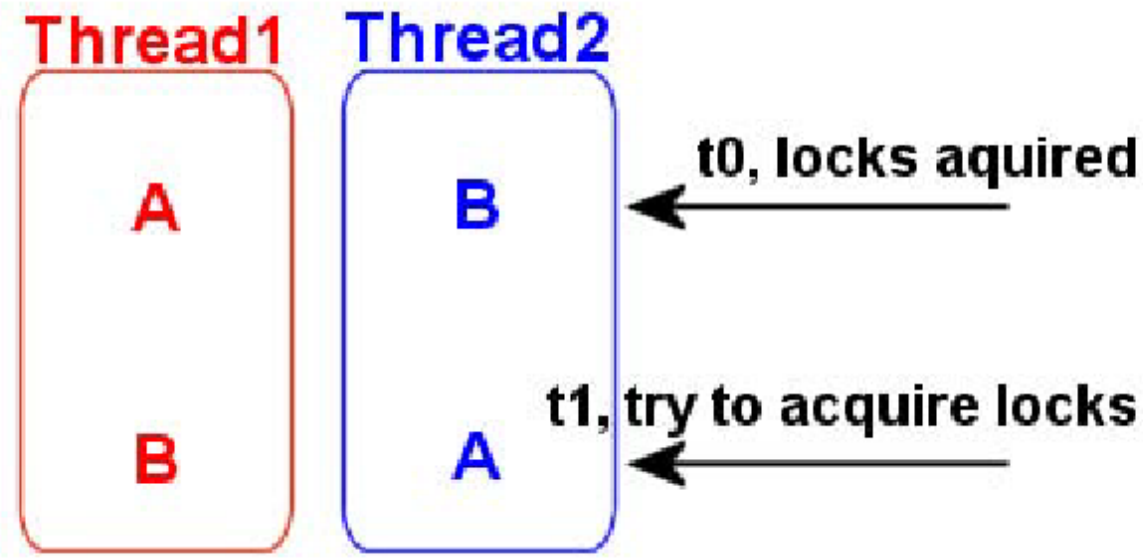
- Option C – is the same as Option B except the bean instance removed from the cache and returned to the pool.
- Option D – is not really a commit option. It is a simple cache invalidation policy. It works the same way as Option A except a background timer removes all instances from the cache.

Pessimistic Locking

The above cache configuration uses a shared instance in the cache. For that reason, a pessimistic lock is used to queue access to the bean. This can cause a lot of contention. The bean is locked to each transaction in turn. The waiting transactions cannot run until the active transaction has committed/rolled back.

Deadlocks

With pessimistic locking deadlocks can occur when beans are accessed in different orders.



Thread 1 holds A and wants B, Thread 2 holds B and wants A. This is impossible. The deadlock is detected by JBoss and the transaction that causes the deadlock is rolled back.

For container-managed transactions (CMT), the exception is caught and the request is retried up to 4 times with delays in between. For BMT, the application can catch the exception and retry the request.

To avoid deadlocks in general, resources should always be allocated in the same order. Using small short-lived transactions helps to avoid deadlocks as does the read-only locks in the following section.

Read Only Lock

The Pessimistic Lock can be relaxed when a method or bean is marked as read-only. The bean is only locked for the duration of the invocation, rather than locking for the whole transaction.

```
<jboss>
<enterprise-beans>
<entity>
<ejb-name>MyEntityBean</ejb-name>
<jndi-name>MyEntityHomeRemote</jndi-name>
<read-only>True</read-only>
</entity>
</enterprise-beans>
</jboss>
```

Listing 4-6, Read-only bean level lock

The whole bean is marked as read only

```
<jboss>
<enterprise-beans>
<entity>
<ejb-name>nextgen.EnterpriseEntity</ejb-name>
<jndi-name>nextgen.EnterpriseEntity</jndi-name>
<method-attributes>
<method>
<method-name>get*</method-name>
<read-only>true</read-only>
</method>
```

Listing 4-7, Read-only method level lock

The methods are marked as read-only.

Instance Per Transaction Configuration

There are cases where a pessimistic lock cannot be used. In this case, each transaction gets its own bean instance. There is no locking within JBoss.

- The database uses repeatable read or serializable isolation. In this case, locks are taken in the database for finder queries. This is not the case with JBoss's pessimistic lock.
- The user wants to use <row-locking> on the database. I.e. select for update. Again, this requires locking on the finder.
- The user wants to use the optimistic locking policy

Cache invalidation

.Cache invalidation allows Commit Option A to be used even though not all modifications are inside the JBoss Server instance or bean deployment. There are a number of variations.

It is important to note that none of the methods mentioned here are not transactional invalidations. There is a delay between the update of data at transaction commit and the invalidation. This makes it unsuitable for Bank Account type data.

The distributed invalidations can be configured to use a JMS Topic or a JGroups multicast. Alternatively, the invalidation can be done directly using JMX to access the cache.

NOTE: The new JBossCache module currently in alpha will include transaction replication of cache information and distributed locks.

- Commit Option D – mentioned above
- Simple clustered invalidation – all access is done through same entity bean deployed on the servers in the same JBoss Cluster. When one server's bean changes the data, it notifies the other machines of the invalidation.
- Read/Write pair – when instance per transaction is required for writes/updates, a separate deployment of the bean can still use commit option A. The two beans are linked by an invalidation bridge. When the write deployment modifies data, it invalidates the read deployment's cache.
- Database Trigger invalidation – a trigger inside the database invalidates the cache.

Chapter 5

5. CMP Optimized Loading

The goal of optimized loading is to load the smallest amount of data required to complete a transaction in the least number of queries. The tuning of JBossCMP depends on a detailed knowledge of the loading process. This chapter describes the internals of the JBossCMP loading process and its configuration. Tuning of the loading process really requires a holistic understanding of the loading system, so this chapter may have to be read more than once.

Loading Scenario

The easiest way to investigate the loading process is to look at a usage scenario. The most common scenario is to locate a collection of entities and iterate over the results performing some operation. The following example generates an html table containing all of the gangsters:

```
public String createGangsterHtmlTable_none() throws FinderException {
   StringBuffer table = new StringBuffer();
   table.append("<table>");

   Collection gangsters = gangsterHome.findAll_none();
   for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
      Gangster gangster = (Gangster)iter.next();
      table.append("<tr>");
      table.append("<td>").append(gangster.getName()).append("</td>");
      table.append("<td>").append(gangster.getNickName()).append("</td>");
      table.append("<td>").append(gangster.getBadness()).append("</td>");
      table.append("</tr>");
   }

   table.append("</table>");
   return table.toString();
}
```

1

2

Listing 5-1, Loading Scenario Example Code

Assume this code is called within a single transaction and all optimized loading has been disabled. At Arrow 1, JBossCMP will execute the following query:

```
SELECT t0_g.id
FROM gangster t0_g
```

```
ORDER BY t0_g.id ASC
```

Listing 5-2, Unoptimized findAll Query

Then at Arrow 2, in order to load the eight Gangsters in the sample database, JBossCMP executes the following eight queries:

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=0)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=1)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=2)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=3)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=4)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=5)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=6)
```

```
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=7)
```

Listing 5-3, Unoptimized Load Queries

There are two problems with this scenario. First, an excessive number of queries are executed because JBossCMP executes one query for findAll and one query for each element found; this is called the "n+1" problem[1] and is addressed with the read-ahead strategies described in the following sections. Second, values of unused fields are loaded because JBossCMP loads the

[1] The reason for this behavior has to do with the handling of query results inside the JBoss container. Although it appears that the actual entity beans selected are returned when a query is executed, JBoss really only returns the primary keys of the matching entities, and does not load the entity until a method is invoked on it.

hangout and organization fields,[2] which are never accessed. Configuration of eager loading is described in the Eager-loading Process section of this chapter. The following table shows the execution of the queries:

Table 5-1, Unoptimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

Load Groups

The configuration and optimization of the loading system begins with the declaration of named load groups in the entity. A load group contains the names of cmp-fields and cmr-fields with a foreign key (e.g., Gangster in the Organization-Gangster example) that will be loaded in a single operation. An example configuration follows:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <load-groups>
            <load-group>
               <load-group-name>basic</load-group-name>
               <field-name>name</field-name>
               <field-name>nickName</field-name>
               <field-name>badness</field-name>
```

[2] Normally JBossCMP would also load the contactInfo field, but for the sake of readability, it has been disabled in this example because contact info maps to seven columns. The actual configuration used to disable the default loading of the contactInfo field is presented in *Listing 5-12*.

```
            </load-group>
            <load-group>
               <load-group-name>contact info</load-group-name>
               <field-name>nickName</field-name>
               <field-name>contactInfo</field-name>
               <field-name>hangout</field-name>
            </load-group>
         </load-groups>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-4, The jbosscmp-jdbc.xml Load Group Declaration

In *Listing 5-4*, two load groups are declared: basic and contact info. Note that the load groups do not need to be mutually exclusive. For example, both of the load groups contain the nickName field. In addition to the declared load groups, JBossCMP automatically adds a group named "*" (the star group) that contains every cmp-field and cmr-field with a foreign key in the entity.

Read-ahead

Optimized loading in JBossCMP is called read-ahead. This term was inherited from JAWS, and refer to the technique of reading the row for an entity being loaded, as well as the next several rows; hence the term read-ahead. JBossCMP implements two main strategies (on-find and on-load) to optimize the loading problem identified in the previous section.

The extra data loaded during read-ahead is not immediately associated with an entity object in memory, as entities are not materialized in JBoss until actually accessed. Instead, it is stored in the preload cache where it remains until it is loaded into an entity or the end of the transaction occurs. The following sections describe the read-ahead strategies.

on-find

The on-find strategy reads additional columns when the query is invoked. If the query in the scenario detailed in *Listing 5-1* is on-find optimized, JBossCMP will execute the following query at Arrow 1:

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

Listing 5-5, on-find Optimized findAll Query

Then at Arrow 2, all of the required data would be in the preload cache, so no additional queries would be executed. This strategy is effective for queries that return a small amount of data, but

becomes very inefficient when trying to load a large result set into memory.[3] The following table shows the execution of this query:

Table 5-2, on-find Optimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

The read-ahead strategy and load-group for a query is defined in the query element. If a read-head strategy is not declared in the query element, the strategy declared in the entity element or defaults element is used. The on-find configuration follows:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <query>
            <query-method>
               <method-name>findAll_onfind</method-name>
               <method-params/>
            </query-method>
            <jboss-ql><![CDATA[
               SELECT OBJECT(g)
               FROM gangster g
               ORDER BY g.gangsterId
            ]]></jboss-ql>
            <read-ahead>
               <strategy>on-find</strategy>
```

[3] JBossCMP uses soft references in the read-ahead cache implementation, so data will be cached and then immediately released.

```
            <page-size>4</page-size>
            <eager-load-group>basic</eager-load-group>
          </read-ahead>
        </query>
      </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-6, The jbosscmp-jdbc.xml on-find Declaration

One problem with the on-find strategy is that it must load additional data for every entity selected. Commonly in web applications only a fixed number of results are rendered on a page. Since the preloaded data is only valid for the length of the transaction, and a transaction is limited to a single web http hit, most of the preloaded data is not used. The on-load strategy discussed in the next section does not suffer from this problem.

on-load

The on-load strategy block loads additional data for several entities when an entity is loaded, starting with the requested entity and the next several entities in the order they were selected.[4] This strategy is based on the theory that the results of a find or ejbSelect will be accessed in forward order. When a query is executed, JBossCMP stores the order of the entities found in the list cache. Later, when one of the entities is loaded, JBossCMP uses this list to determine the block of entities to load. The number of lists stored in the cache is specified with the list-cache-max element of the entity. This strategy is also used when faulting in data not loaded in the on-find strategy. With this strategy, the query executed at Arrow 1 remains unchanged.

```
SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

Listing 5-7, on-load (Unoptimized) findAll Query

If, for example, the page size is set to four, JBossCMP will execute the following two queries to load the name, nickName and badness fields for the entities:

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
```

[4] This is the read-ahead technique from JAWS.

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=4) OR (id=5) OR (id=6) OR (id=7)
```

Listing 5-8, on-load Optimized Load Queries

The following table shows the execution of these queries:

Table 5-3, on-load Optimized Query Execution

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

As with the on-find strategy, on-load is declared in the read-ahead element. The on-load configuration follows:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <query>
            <query-method>
               <method-name>findAll_onload</method-name>
               <method-params/>
            </query-method>
            <jboss-ql><![CDATA[
               SELECT OBJECT(g)
               FROM gangster g
               ORDER BY g.gangsterId
            ]]></jboss-ql>
            <read-ahead>
```

```
            <strategy>on-load</strategy>
            <page-size>4</page-size>
            <eager-load-group>basic</eager-load-group>
         </read-ahead>
      </query>
   </entity>
</enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-9, The jbosscmp-jdbc.xml on-load Declaration

none

The none strategy is really an anti-strategy. This strategy causes the system to fall back to the default lazy-load code, and specifically does not read-ahead any data or remember the order of the found entities. This results in the queries and performance shown at the beginning of this chapter. The none strategy is declared with a read-ahead element. If the read-ahead element contains a page-size element or eager-load-group, it is ignored. The none strategy is declared in the following configuration:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <query>
            <query-method>
               <method-name>findAll_none</method-name>
               <method-params/>
            </query-method>
            <jboss-ql><![CDATA[
               SELECT OBJECT(g)
               FROM gangster g
               ORDER BY g.gangsterId
            ]]></jboss-ql>
            <read-ahead>
               <strategy>none</strategy>
            </read-ahead>
         </query>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-10, The jbosscmp-jdbc.xml none Declaration

Loading Process

In the previous section several steps use the phrase "when the entity is loaded." This was intentionally left vague because the commit option specified for the entity and the current state of the transaction determine when an entity is loaded. The following section describes the commit options and the loading processes.

Commit Options

Central to the loading process are the commit options, which control when the data for an entity expires. JBoss supports four commit options A, B, C and D. The first three are described in section 10.5.9 of the **Enterprise JavaBeans Specification Version 2.0 Final Release** and the forth, D, is specific to JBoss. A detailed description of each commit option follows:

- **A**: JBossCMP assumes it is the sole user of the database; therefore, JBossCMP can cache the current value of an entity between transactions, which can result is substantial performance gains. As a result of this assumption, no data managed by JBossCMP can be changed outside of JBossCMP. For example, changing data in another program or with the use of direct JDBC (even within JBoss) will result in an inconsistent database state.

- **B**: JBossCMP assumes that there is more than one user of the database but keeps the context information about entities between transactions. This context information is used for optimizing loading of the entity.[5] This is the default commit option.

- **C**: JBossCMP discards all entity context information at the end of the transaction.

- **D**: This is a JBoss specific commit option. This option is similar to commit option A, except that the data only remains valid for a specified amount of time.

The commit option is declared in the jboss.xml file. For a detailed description of this file see the **JBoss 3.0 Quick Start Guide**. The following example changes the commit option to A for all entity beans in the application:

```
<jboss>
   <container-configurations>
      <container-configuration>
         <container-name>Standard CMP 2.x EntityBean</container-name>
         <commit-option>A</commit-option>
      </container-configuration>
   </container-configurations>
</jboss>
```

Listing 5-11, The jboss.xml Commit Option Declaration

Eager-loading Process

One of the most important changes in CMP 2.0 is the change from using class fields for cmp-fields to abstract accessor methods. In CMP 1.x, the container could not know which fields were required in a transaction, so the container had to eager-load every field when loading the

[5] In a future version, JBossCMP will be able to keep the current data of a commit option B entity between transactions and validate that the data is still current using last-update optimistic locking. For entities that contain a large amount of data, this will result in a significant performance enhancement.

bean. In CMP 2.x, the container creates the implementation for the abstract accessors, so the container can know when the data for a field is required. JBossCMP can be configured to eager-load only some of the fields when loading an entity, and later lazy-load the remaining fields as needed.

When an entity is loaded, JBossCMP must determine the fields that need to be loaded. By default, JBossCMP will use the eager-load-group of the last query that selected this entity. If the entity has not been selected in a query, or the last query used the none read-ahead strategy, JBossCMP uses the default eager-load-group declared for the entity. In the following configuration, the basic load group is set as the default eager-load-group for the GangsterEJB entity:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <load-groups>
            <load-group>
               <load-group-name>most</load-group-name>
               <field-name>name</field-name>
               <field-name>nickName</field-name>
               <field-name>badness</field-name>
               <field-name>hangout</field-name>
               <field-name>organization</field-name>
            </load-group>
         </load-groups>
         <eager-load-group>most</eager-load-group>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-12, The jbosscmp-jdbc.xml Eager Load Declaration

The eager loading process is initiated the first time a method is called on an entity in a transaction. A detailed description of the load process follows:

1. If the entity context is still valid, no loading is necessary, and therefore the loading process is done. The entity context will be valid when using commit option A, or when using commit option D, and the data has not timed out.

2. Any residual data in the entity context is flushed. This assures that old data does not bleed into the new load.

3. The primary key value is injected back into the primary key fields. The primary key object is actually independent of the fields and needs to be reloaded after the flush in step 2.

4. All data in the preload cache for this entity is loaded into the fields.

5. JBossCMP determines the additional fields that still need to be loaded. Normally the fields to load are determined by the eager-load group of the entity, but can be overridden if the entity was located using a query or cmr-field with an on-find or on-load read-ahead strategy. If all of the fields have already been loaded, the load process skips to step 7.

6. A query is executed to select the necessary column. If this entity is using the on-load strategy, a page of data is loaded as described in the on-load section. The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.

7. The ejbLoad method of the entity is called.

Lazy-loading Process

Lazy-loading is the other half of eager-loading. If a field is not eager-loaded, it must be lazy-loaded. When the bean accesses an unloaded field, JBossCMP loads the field and any field in a lazy-load-group of which the unloaded field is a member. JBossCMP performs a set join and then removes any field that is already loaded. An example follows:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>GangsterEJB</ejb-name>
         <load-groups>
            <load-group>
               <load-group-name>basic</load-group-name>
               <field-name>name</field-name>
               <field-name>nickName</field-name>
               <field-name>badness</field-name>
            </load-group>
            <load-group>
               <load-group-name>contact info</load-group-name>
               <field-name>nickName</field-name>
               <field-name>contactInfo</field-name>
               <field-name>hangout</field-name>
            </load-group>
         </load-groups>
         <lazy-load-groups>
            <load-group-name>basic</load-group-name>
            <load-group-name>contact info</load-group-name>
         </lazy-load-groups>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Listing 5-13, The jbosscmp-jdbc.xml Lazy Load Group Declaration

When the bean provider calls getName() with this configuration, JBossCMP loads name, nickName and badness (assuming they are not already loaded). When the bean provider calls getNickName(), the name, nickName, badness, contactInfo, and hangout are loaded. A detailed description of the lazy-loading process follows:

1. All data in the preload cache for this entity is loaded into the fields.
2. If the field value was loaded by the preload cache the lazy-load process is finished.
3. JBossCMP finds all of the lazy load groups that contain this field, performs a set join on the groups, and removes any field that has already been loaded.
4. A query is executed to select the necessary column. As in the basic load process, JBossCMP may load a block of entities. The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.

Relationships

Relationships are a special case in lazy-loading because a cmr-field is both a field and query. As a field it can be on-load block loaded, meaning the value of the currently sought entity and the values of the cmr-field for the next several entities are loaded. As a query, the field values of the related entity can be preloaded on-find.

Again, the easiest way to investigate the loading is to look at a usage scenario. In this example, an html table is generated containing each gangster and their hangout. The example code follows:

```
public String createGangsterHangoutHtmlTable() throws FinderException {
   StringBuffer table = new StringBuffer();
   table.append("<table>");

   Collection gangsters = gangsterHome.findAll_onfind();
   for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
      Gangster gangster = (Gangster)iter.next();
      Location hangout = gangster.getHangout();
      table.append("<tr>");
      table.append("<td>").append(gangster.getName()).append("</td>");
      table.append("<td>").append(gangster.getNickName()).append("</td>");
      table.append("<td>").append(gangster.getBadness()).append("</td>");
      table.append("<td>").append(hangout.getCity()).append("</td>");
      table.append("<td>").append(hangout.getState()).append("</td>");
      table.append("<td>").append(hangout.getZipCode()).append("</td>");
      table.append("</tr>");
   }

   table.append("</table>");
   return table.toString();
}
```

Listing 5-14, Relationship Lazy Loading Example Code

For this example, the configuration of the Gangster findAll_onfind query is unchanged from the on-find section. The configuration of the Location entity and Gangster-Hangout relationship follows:

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>LocationEJB</ejb-name>
         <load-groups>
            <load-group>
               <load-group-name>quick info</load-group-name>
               <field-name>city</field-name>
               <field-name>state</field-name>
               <field-name>zipCode</field-name>
            </load-group>
         </load-groups>
         <eager-load-group/>
      </entity>
   </enterprise-beans>
   <relationships>
      <ejb-relation>
         <ejb-relation-name>Gangster-Hangout</ejb-relation-name>
         <foreign-key-mapping/>
         <ejb-relationship-role>
            <!-- outdented to fit on a printed page -->
<ejb-relationship-role-name>gangster-has-a-hangout</ejb-relationship-role-name>
            <key-fields/>
            <read-ahead>
               <strategy>on-find</strategy>
```

```
            <page-size>4</page-size>
            <eager-load-group>quick info</eager-load-group>
         </read-ahead>
      </ejb-relationship-role>
      <ejb-relationship-role>
         <!-- outdented to fit on a printed page -->
<ejb-relationship-role-name>hangout-for-a-gangster</ejb-relationship-role-name>
         <key-fields>
            <key-field>
               <field-name>locationId</field-name>
               <column-name>hangout</column-name>
            </key-field>
         </key-fields>
      </ejb-relationship-role>
   </ejb-relation>
</relationships>
</jbosscmp-jdbc>
```

Listing 5-15, The jbosscmp-jdbc.xml Relationship Lazy Loading Configuration

At Arrow 1, JBossCMP will execute the following query:

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

Listing 5-16, on-find Optimized findAll Query

Then at Arrow 2, JBossCMP executes the following two queries to load the city, state, and zip fields of the hideout:

```
SELECT gangster.id, gangster.hangout,
      location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=0) OR (gangster.id=1) OR
       (gangster.id=2) OR (gangster.id=3))
SELECT gangster.id, gangster.hangout,
      location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
      ((gangster.id=4) OR (gangster.id=5) OR
       (gangster.id=6) OR (gangster.id=7))
```

Listing 5-17, on-find Optimized Relationship Load Queries

The following table shows the execution of the queries:

Table 5-4, on-find Optimized Relationship Query Execution

Gangster					Location			
id	name	nick_name	badness	hangout	id	city	st	zip
0	Yojimbo	Bodyguard	7	0	0	San Fran	CA	94108
1	Takeshi	Master	10	1	1	San Fran	CA	94133
2	Yuriko	Four finger	4	2	2	San Fran	CA	94133
3	Chow	Killer	9	3	3	San Fran	CA	94133
4	Shogi	Lightning	8	4	4	San Fran	CA	94133
5	Valentino	Pizza-Face	4	5	5	New York	NY	10017
6	Toni	Toothless	2	6	6	Chicago	IL	60661
7	Corleone	Godfather	6	7	7	Las Vegas	NV	89109

Transactions

All of the examples presented in this chapter have been defined to run in a transaction. Transaction granularity is a dominating factor in optimized loading because transactions define the lifetime of preloaded data. If the transaction completes, commits, or rolls back, the data in the preload cache is lost. This can result in a severe negative performance impact.

The performance impact of running without a transaction will be demonstrated with an example similar to *Listing 5-1*. This example uses an on-find optimized query that selects the first four gangsters (to keep the result set small), and it is executed without a wrapper transaction. The example code follows:

```
public String createGangsterHtmlTable_no_tx() throws FinderException {
   StringBuffer table = new StringBuffer();
   table.append("<table>");
   Collection gangsters = gangsterHome.findFour();
   for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
      Gangster gangster = (Gangster)iter.next();
      table.append("<tr>");
      table.append("<td>").append(gangster.getName()).append("</td>");
      table.append("<td>").append(gangster.getNickName()).append("</td>");
      table.append("<td>").append(gangster.getBadness()).append("</td>");
      table.append("</tr>");
   }
   table.append("</table>");
   return table.toString();
}
```

1

2

Listing 5-18, No Transaction Loading Example Code

The query executed at Arrow 1 follows:

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
WHERE t0_g.id < 4
ORDER BY t0_g.id ASC
```

Listing 5-19, No Transaction on-find Optimized findAll Query

Normally this would be the only query executed, but since this code is not running in a transaction, all of the preloaded data is thrown away as soon as findAll returns. Then at Arrow 2 JBossCMP executes the following four queries (one for each loop):[6]

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
```

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=1) OR (id=2) OR (id=3)
```

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=2) OR (id=3)
```

[6] It's actually worse than this. JBossCMP executes each of these queries three times; once for each cmp-field that is accessed. This is because the preloaded values are discarded between the cmp-field accessor calls.

```
SELECT name, nick_name, badness
FROM gangster
WHERE (id=3)
```

Listing 5-20, No Transaction on-load Optimized Load Queries

id	name	nick_name	badness
0	Yojimbo	Bodyguard	7
1	Takeshi	Master	10
2	Yuriko	Four finger	4
3	Chow	Killer	9

The following table shows the execution of the queries:

$$n+(n-1)+(n-2)+(n-3)+...+1=\frac{n(n+1)}{2}=O(n^2)$$

Table 5-5, No Transaction on-find Optimized Query Execution

This performance is much worse than read-ahead none because of the amount of data loaded from the database. The number of rows loaded is determined by the following equation:

This all happens because the transaction in the example is bounded by a single call on the entity. This brings up the important question "How do I run my code in a transaction?" The answer depends on where the code runs. If it runs in an EJB (session, entity, or message driven), the method must be marked with the Required or RequiresNew trans-attribute in the assembly-descriptor. If the code is not running in an EJB, a user transaction is necessary. The following code wraps a call to the method declared in *Listing 5-18* with a user transaction:

```
public String createGangsterHtmlTable_with_tx() throws FinderException {
   UserTransaction tx = null;
   try {
      InitialContext ctx = new InitialContext();
      tx = (UserTransaction) ctx.lookup("UserTransaction");
      tx.begin();
```

```
      String table = createGangsterHtmlTable_no_tx();

      if(tx.getStatus() == Status.STATUS_ACTIVE) {
         tx.commit();
      }
      return table;
   } catch(Exception e) {
      try {
         if(tx != null) tx.rollback();
      } catch(SystemException unused) {
         // eat the exception we are exceptioning out anyway
      }
      if(e instanceof FinderException) {
         throw (FinderException) e;
      }
      if(e instanceof RuntimeException) {
         throw (RuntimeException) e;
      }
      throw new EJBException(e);
   }
}
```

Listing 5-21, User Transaction Example Code

Chapter

6

6. WEB Containers

The JBossWeb container can be found in a deploy/jbossweb-xxxx.sar where xxxx depends upon the embedded web container configuration. This chapter deals with the Tomcat4.1 version.

Connectors

The connection configurations can be found in META-INF/jboss-service.xml

The connectors are pools of threads that accept inbound requests and process them

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
   address="${jboss.bind.address}" port="8080" minProcessors="5" maxProcessors="100"
   enableLookups="true" acceptCount="10" debug="0"
   connectionTimeout="20000" useURIValidationHack="false"/>
```

Listing 6-1, Tomcat connector configuration

- acceptCount – controls the length of the queue of waiting processes when there are no processors available
- bufferSize – controls the size of the input stream used to read data from the socket. The default is 2K
- connectionTimeout – how long to wait before a URI is received from the stream the default is 20 seconds. This avoid problems where a client opens a connection and does not send any data
- maxProcessors – The maximum number of threads in the pool, this is a strict pool
- minProcessors – Threads in the pool constructed at startup
- tcpNoDelay – set to true when data should be sent to the client without waiting for the buffer to be full. This reduces latency at the cost of more packets being sent over the network (default true)

- enableLookups – whether to perform a reverse dns lookup to prevent snoofing this can cause problems when a dns is misbehaving. It can be turned off when you implicitly trust all clients
- compression – whether to compress data sent to the client, valid values are "off", "on", "force" or a numerical value that specifies the number of bytes output before compression is used. The default is "off"
- connectionLinger – how long connections should linger after they are closed. The default is -1 (no linger)
- disableUpdloadTimeout – whether to avoid timeouts while the client is uploading data to the servlet, the default is false

JSP Optimizations

The jasper servlet configuration can be found in web.xml

The jasper servlet by is configured in development mode by default. This means the jsp page is checked for modification on every access.

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>logVerbosityLevel</param-name>
        <param-value>WARNING</param-value>
    </init-param>
    <init-param>
        <param-name>development</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>
```

Listing 6-2, Jasper configuration

- development - turns off checking for changed jsps

- checkInterval – if you still want jsps to be check it can be done on a timer rather than every access. Development must be true for this to work.

Precompilation of jsps

Jsps can be precompiled as servlets to avoid compilation on first access. Details can be found at http://www.jakarta.apache.org/tomcat/tomcat-4.1-doc/jasper-how-to

The tomcat jars mentioned in the ant definition should be replaced with the jboss jars and sars in lib/ server/default/lib and server/default/deploy plus your own jars if they have not been included in the JBoss directories.

Chapter

7

7. The Front End

This section covers remote invocations.

RMI

As already mentioned this relies on Virtual machine configuration

Pooled Invoker

An optimized invoker that does not really on the VM. It is configured in conf/jboss-service.xml

```
   <mbean code="org.jboss.invocation.pooled.server.PooledInvoker"

      name="jboss:service=invoker,type=pooled">

      <attribute name="NumAcceptThreads">1</attribute>

      <attribute name="MaxPoolSize">300</attribute>

      <attribute name="ClientMaxPoolSize">300</attribute>

      <attribute name="SocketTimeout">60000</attribute>

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

      <attribute name="ServerBindPort">4445</attribute>

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

      <attribute name="ClientConnectPort">0</attribute>

      <attribute name="EnableTcpNoDelay">false</attribute>



      <depends optional-attribute-
name="TransactionManagerService">jboss:service=TransactionManager</depends>

   </mbean>
```

Listing 7-1, Pooled Invoker

- NumAcceptThreads – The controlling threads used to accept connections from the client
- MaxPoolSize – A strict pool of threads to service requests on the server

- ClientMaxPoolSize – A strict pool of threads to service requests on the client
- Backlog – the number of requests in the queue when all the processing threads are in use
- EnableTcpDelay – whether information should be sent before the buffer is full

IIOP Configuration

The RMI/IIOP transport is handled by jacorb. The configuration can be found in conf/jacorb.properties in the all configuration. The property file is internally documented.

HTTP Invoker

This invoker uses the web container. See the web container for configuration. If you use this invoker, do not forget that the web container is being used for normal web requests and remote invocations.

Chapter

8

8. The Back End

This section covers connection pooling including datasource pooling.

Generic Pool configuration

This example shows a datasource. The pooling parameters apply to all resource adapter deployments.

```
<datasources>
  <local-tx-datasource>
    <jndi-name>GenericDS</jndi-name>
    <connection-url>[jdbc: url for use with Driver class]</connection-url>
    <driver-class>[fully qualified class name of java.sql.Driver implementation]</driver-class>
    <user-name>x</user-name>
    <password>y</password>
    <!-- you can include connection properties that will get passed in
     the DriverManager.getConnection(props) call-->
    <!-- look at your Driver docs to see what these might be -->
    <config-property name="SomeProperty" type="java.lang.String">x</config-property>
    <connection-property name="char.encoding">UTF-8</connection-property>
    <transaction-isolation>TRANSACTION_SERIALIZABLE</transaction-isolation>
    <!--pooling parameters-->
    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>
    <blocking-timeout-millis>5000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
    <prepared-statement-cache-size>100</prepared-statement-cache-size>
```

Listing 8-1, Pooling Parameters

- min-pool-size – the minimum number of connections in the pool (the pool is lazily constructed, i.e. it will be empty until first access. One used it will always have at least the min-pool-size connections)
- max-pool-size – the strict maximum size of the pool
- blocking-time-millis – the time to wait for a connection when they are all in use
- idle-timout-minutes – the length of time an idle connection will remain in the pool before it used.

Resource adapter parameters

- config-property - each resource adapter can have its own configuration properties

JDBC Connections

These parameters are specific to JBoss's jdbc resource adapter

JDBC Connection parameters

- connection-property – parameters passed to each connection

JBoss specific jdbc properties

- prepared-statement-cache-size – the number of prepared statements per connection in the cache (the default is 0 meaning no prepared statement cache).

Cached Connection Manager

The cached connection manager configuration can be found in deploy/transaction-service.xml

```
<server>


  <!--

     | The CachedConnectionManager is used partly to relay started UserTransactions to

     | open connections so they may be enrolled in the new tx.

   -->

  <mbean code="org.jboss.resource.connectionmanager.CachedConnectionManager"

         name="jboss.jca:service=CachedConnectionManager">

    <depends optional-attribute-
name="TransactionManagerServiceName">jboss:service=TransactionManager</depends>
```

```
    <!--SpecCompliant false means JBoss will close connections left open when you
        return from a method call and generate a loud warning.  SpecCompliant true
        means JBoss will disconnect connection handles left open on return from a
        method call and reconnect them with an appropriate (security, tx)
        connection on the next call to the same object.-->
    <attribute name="SpecCompliant">false</attribute>

    <!-- Enable connection close debug monitoring -->
    <attribute name="Debug">false</attribute>

  </mbean>

</server>
```

Listing 8-2, Cached Connection Manager

- Debug – this is useful during development. It helps catch unclosed connections that would otherwise cause leaks in the connection pools. However, it is a global point of contention that should be turned off (false) in production or stress testing.

Chapter

9

9. Miscellaneous

This section covers configurations not covered by other sections.

JNDI

It is a common mistake to configure a provider url on the server

`java.naming.provider.url=jnp://localhost:1099`

This forces jndi to use local sockets when the server its own jndi tree. Inside the server, jndi should be done using method calls. This occurs when there is no provider url.

This should be contrasted with the client, where the lack of provider url forces a multicast discovery for a HAJNDI instance on the network unless jnp.disableDiscovery=true is specified.

Transaction Manager

The transaction manager can be found in conf/jboss-service.xml

The transaction timeout helps to break blocked threads that are stuck in a loop, accessing an external resource or waiting too long for a locked object.

```
<!--
   | The fast in-memory transaction manager.
 -->
<mbean code="org.jboss.tm.TransactionManagerService"
   name="jboss:service=TransactionManager"
   xmbean-dd="resource:xmdesc/TransactionManagerService-xmbean.xml">
   <attribute name="TransactionTimeout">300</attribute>
   <depends optional-attribute-name="XidFactory">jboss:service=XidFactory</depends>
</mbean>
```

Listing 9-1, Transaction Manager

- TransactionTimeout – controls how long a transaction lasts before the transaction is marked for rollback and the thread is interrupted. The default is 5 minutes.

JMS Invocation Layer

The jms invocation layers are used to access the jms server remotely. When running inside the jms server, method calls should be used. This is done through the connection factory java:/ConnectionFactory. Even better is java:/JmsXA which provides pooled transaction access to jms. If transactions are not required it can be deployed as a <no-tx-connection-factory>.

The following shows UIL2 which is the recommend remote invocation layer it can be found in deploy/jms/uil2-service.xml

```
<mbean code="org.jboss.mq.il.uil2.UILServerILService"

      name="jboss.mq:service=InvocationLayer,type=UIL2">

  <depends optional-attribute-name="Invoker">jboss.mq:service=Invoker</depends>

  <attribute name="ConnectionFactoryJNDIRef">UIL2ConnectionFactory</attribute>

  <attribute name="XAConnectionFactoryJNDIRef">UIL2XAConnectionFactory</attribute>

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

  <attribute name="ServerBindPort">8093</attribute>

  <attribute name="PingPeriod">60000</attribute>

  <attribute name="EnableTcpNoDelay">true</attribute>

  <!-- Used to disconnect the client if there is no activity -->

  <!-- Ensure this is greater than the ping period -->

  <attribute name="ReadTimeout">70000</attribute>

  <!-- The size of the buffer (in bytes) wrapping the socket -->

  <!-- The buffer is flushed after each request -->

  <attribute name="BufferSize">2048</attribute>

  <!-- Large messages may block the ping/pong -->

  <!-- A pong is simulated after each chunk (in bytes) for both reading and writing -->

  <!-- It must be larger than the buffer size -->

  <attribute name="ChunkSize">1000000</attribute>

</mbean>
```

Listing 9-2, JMS Invocation Layer

- PingPeriod – the length of time between pings by the client to test the connection is still alive

- ReadTimeout – the maximum time no activity is allowed on the socket before the server will disconnect the client
- EnableTcpNoDelay – whether data should be sent before the buffer is full
- BufferSize – the size of the buffer when sending and reeiving data, the buffer is always flushed for each request.

JMS Message Cache

The message cache is used to protect memory when memory is full. When memory passes the high memory mark, the cache starts pushing messages to disk to keep memory below the max memory mark. This only works when jms messages are actually using the memory. The parameters should be close the memory allocated to the virtual machine's heap, but some margin for error should be given in case a large message arrives when memory is full. The following are the default settings assuming the heap is 64M

It is configured along with the persistent manager, the default is deploy/jms/hsqldb-jdbc2-service.xml

```
<!--
   | The MessageCache decides where to put JBossMQ message that
   | are sitting around waiting to be consumed by a client.
   | The memory marks are in Megabytes.  Once the JVM memory usage hits
   | the high memory mark, the old messages in the cache will start getting
   | stored in the DataDirectory.  As memory usage gets closer to the
   | Max memory mark, the amount of message kept in the memory cache aproaches 0.
 -->
<mbean code="org.jboss.mq.server.MessageCache"
       name="jboss.mq:service=MessageCache">
  <attribute name="HighMemoryMark">50</attribute>
  <attribute name="MaxMemoryMark">60</attribute>
  <attribute name="CacheStore">jboss.mq:service=PersistenceManager</attribute>
</mbean>
```

Listing 9-3, JMS Message Cache

Appendix

A

A. About The JBoss Group

The JBoss Group, LLC is an Atlanta-based professional services company created by Marc Fleury, founder and lead developer of the JBoss J2EE-based open source web application server. The JBoss Group brings together core JBoss developers to provide services such as training, support and consulting, as well as management of the JBoss software and services affiliate programs. These commercial activities subsidize the development of the free core JBoss server. For additional information on The JBoss Group, see the JBoss website at **http://www.jboss.org/jbossgroup/services.jsp**.

Appendix

B

B. Changes to This Document

This page lists the changes made to this document. When editing, please make sure you have a change tracker turned on MS Word -> Tools -> Track Changes -> Highlight Changes

23 January 2004: Added additional comments to fine tuning garbage collection [JPL]