Chapter No. 7
"JBoss AS Cluster Tuning"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.7 "JBoss AS Cluster Tuning"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Francesco Marchioni is a Sun Certified Enterprise Architect employed by an Italian company based in Rome. He started learning Java in 1997 and since then he has followed the path to the newest Application Program Interfaces released by Sun. Since 2000, he has joined the JBoss Community when the application server was running release 2.X.

He has spent many years as a software consultant, where he has created many successful software migrations from vendor platforms to the open source products like JBoss AS, fulfilling the tight budget requirements of current times.

In the past five years he has started authoring technical articles for O'Reilly Media and running an IT portal focused on JBoss products (http://www.mastertheboss.com).


For More Information:
I'd like to thank Packt Publishing for offering another opportunity to write about my favorite application server. I'd also like to thank my family for supporting me during the creation of this book, in particular my father for teaching me all the tricks of the glorious C-64 when I was a kid.

Also a special thanks to the people that collaborated to this book: my friend Luca Viola who lent me the hardware for these benchmarks. JBoss guys Carlo de Wolf and Alessio Soldano for their precious advices. And last but not least the technical reviewers - Devon Hillard and Kushal Paudyal—and all the Packt Pub staff for their competent cooperation.

For More Information:  
JBoss AS 5 Performance Tuning

JBoss AS 5 Performance Tuning will teach you how to deliver fast applications on the JBoss Application Server and Apache Tomcat, giving you a decisive competitive advantage over your competitors. You will learn how to optimize hardware resources, meeting your application requirements with less expenditure.

The performance of Java Enterprise applications is the sum of a set of components including the Java Virtual Machine configuration, the application server configuration (in our case, JBoss AS), the application code itself, and ultimately the operating system. This book will show you how to apply the correct tuning methodology and use the tuning tools that will help you to monitor and address any performance issues.

By looking more closely at the Java Virtual Machine, you will get a deeper understanding of what the available options are for your applications, and how their performance will be affected. Learn about thread pool tuning, EJB tuning, and JMS tuning, which are crucial parts of enterprise applications.

The persistence layer and the JBoss Clustering service are two of the most crucial elements which need to be configured correctly in order to run a fast application. These aspects are covered in detail with a chapter dedicated to each of them.

Finally, Web server tuning is the last (but not least) topic covered, which shows how to configure and develop web applications that get the most out of the embedded Tomcat web server.

For More Information:
What This Book Covers

Chapter 1, Performance Tuning Concepts, discusses correct tuning methodology and how it fits in the overall software development cycle.

Chapter 2, Installing the Tools for Tuning, shows how to install and configure the instruments for tuning, including VisualVM, JMeter, Eclipse TPTP Platform, and basic OS tools.

Chapter 3, Tuning the Java Virtual Machine, provides an in-depth analysis of the JVM heap and garbage collector parameters, which are used to start up the application server.

Chapter 4, Tuning the JBoss AS, discusses the application server's core services including the JBoss System Thread Pool, the Connection Pool, and the Logging Service.

Chapter 5, Tuning the Middleware Services, covers the tuning of middleware services including the EJB and JMS services.

Chapter 6, Tuning the Persistence Layer, introduces the principles of good database design and the core concepts of Java Persistence API with special focus on JBoss's implementation (Hibernate).

Chapter 7, JBoss AS Cluster Tuning, covers JBoss Clustering service covering the low-level details of server communication and how to use JBoss Cache for optimal data replication and caching.

Chapter 8, Tomcat Web Server Tuning, covers the JBoss Web server performance tuning including mod_jk, mod_proxy, and mod_cluster modules.

Chapter 9, Tuning Web Applications on JBoss AS, discusses developing fast web applications using JSF API and JBoss richfaces libraries.

For More Information:
6th Circle of Hell: Heresy. This circle houses administrators who accurately set up a cluster to use Buddy Replication. Without caring about steady sessions.

Clustering allows us to run applications on several parallel instances (also known as cluster nodes). The load is distributed across different servers, and even if any of the servers fails, the application is still accessible via other cluster nodes. Clustering is crucial for scalable Enterprise applications, as you can improve performance by simply adding more nodes to the cluster.

In this chapter, we will cover the basic building blocks of JBoss Clustering with the following schedule:

- A short introduction to JBoss Clustering platform
- In the next section we will cover the low level details of the JGroups library, which is used for all clustering-related communications between nodes
- In the third section we will discuss JBoss Cache, which provides distributed cache and state replication services for the JBoss cluster on top of the JGroups library

Introduction to JBoss clustering

Clustering plays an important role in Enterprise applications as it lets you split the load of your application across several nodes, granting robustness to your applications. As we discussed earlier, for optimal results it’s better to limit the size of your JVM to a maximum of 2-2.5GB, otherwise the dynamics of the garbage collector will decrease your application’s performance.

Combining relatively smaller Java heaps with a solid clustering configuration can lead to a better, scalable configuration plus significant hardware savings.

For More Information:
The only drawback to scaling out your applications is an increased complexity in the programming model, which needs to be correctly understood by aspiring architects.

JBoss AS comes out of the box with clustering support. There is no all-in-one library that deals with clustering but rather a set of libraries, which cover different kinds of aspects. The following picture shows how these libraries are arranged:

![Diagram showing the arrangement of libraries](image)

The backbone of JBoss Clustering is the JGroups library, which provides the communication between members of the cluster. Built upon JGroups we meet two building blocks, the JBoss Cache framework and the HAPartition service.

JBoss Cache handles the consistency of your application across the cluster by means of a replicated and transactional cache.

On the other hand, HAPartition is an abstraction built on top of a JGroups Channel that provides support for making and receiving RPC invocations from one or more cluster members. For example HA-JNDI (High Availability JNDI) or HA Singleton (High Availability Singleton) both use HAPartition to share a single Channel and multiplex RPC invocations over it, eliminating the configuration complexity and runtime overhead of having each service create its own Channel. (If you need more information about the HAPartition service you can consult the JBoss AS documentation [here](http://community.jboss.org/wiki/jBossAS5ClusteringGuide).)

In the next section we will learn more about the JGroups library and how to configure it to reach the best performance for clustering communication.

For More Information:
Configuring JGroups transport

Clustering requires communication between nodes to synchronize the state of running applications or to notify changes in the cluster definition.

JGroups (http://jgroups.org/manual/html/index.html) is a reliable group communication toolkit written entirely in Java. It is based on IP multicast, but extends by providing reliability and group membership.

Member processes of a group can be located on the same host, within the same Local Area Network (LAN), or across a Wide Area Network (WAN). A member can be in turn part of multiple groups.

The following picture illustrates a detailed view of JGroups architecture:

A JGroups process consists basically of three parts, namely the Channel, Building blocks, and the Protocol stack. The Channel is a simple socket-like interface used by application programmers to build reliable group communication applications. Building blocks are an abstraction interface layered on top of Channels, which can be used instead of Channels whenever a higher-level interface is required. Finally we have the Protocol stack, which implements the properties specified for a given channel.
In theory, you could configure every service to bind to a different Channel. However this would require a complex thread infrastructure with too many thread context switches. For this reason, JBoss AS is configured by default to use a single Channel to multiplex all the traffic across the cluster.

The Protocol stack contains a number of layers in a bi-directional list. All messages sent and received over the channel have to pass through all protocols. Every layer may modify, reorder, pass or drop a message, or add a header to a message. A fragmentation layer might break up a message into several smaller messages, adding a header with an ID to each fragment, and re-assemble the fragments on the receiver's side.

The composition of the Protocol stack (that is, its layers) is determined by the creator of the channel: an XML file defines the layers to be used (and the parameters for each layer).

Knowledge about the Protocol stack is not necessary when just using Channels in an application. However, when an application wishes to ignore the default properties for a Protocol stack, and configure their own stack, then knowledge about what the individual layers are supposed to do is needed.

In JBoss AS, the configuration of the Protocol stack is located in the file, `<server>\deploy\cluster\jgroups-channelfactory.sar\META-INF\jgroups-channelfactory-stacks.xml`. The file is quite large to fit here, however, in a nutshell, it contains the following basic elements:

For More Information:
The first part of the file includes the UDP transport configuration. UDP is the default protocol for JGroups and uses multicast (or, if not available, multiple unicast messages) to send and receive messages.

For More Information:

Next to the UDP transport configuration, three protocol stacks are defined:

- **udp**: The default IP multicast based stack, with flow control
- **udp-async**: The protocol stack optimized for high-volume asynchronous RPCs
- **udp-sync**: The stack optimized for low-volume synchronous RPCs

A multicast UDP socket can send and receive datagrams from multiple clients. The interesting and useful feature of multicast is that a client can contact multiple servers with a single packet, without knowing the specific IP address of any of the hosts.
Thereafter, the **TCP transport configuration** is defined. TCP stacks are typically used when IP multicasting cannot be used in a network (for example, because it is disabled) or because you want to create a network over a WAN (that's conceivably possible but sharing data across remote geographical sites is a scary option from the performance point of view).

You can opt for two TCP protocol stacks:

- **tcp**: Addresses the default TCP Protocol stack which is best suited to high-volume asynchronous calls.
- **tcp-async**: Addresses the TCP Protocol stack which can be used for low-volume synchronous calls.

If you need to switch to TCP stack, you can simply include the following in your command line args that you pass to JBoss:

```
-Djboss.default.jgroups.stack=tcp
```

Since you are not using multicast in your TCP communication, this requires configuring the addresses/ports of all the possible nodes in the cluster. You can do this by using the property `-Djgroups.tcpping.initial_hosts`. For example:

```
-Djgroups.tcpping.initial_hosts=host1[7600],host2[7600]
```

Ultimately, the configuration file contains two stacks which can be used for optimising JBoss Messaging Control Channel (`jbm-control`) and Data Channel (`jbm-data`).

### How to optimize the UDP transport configuration

The default UDP transport configuration ships with a list of attributes, which can be tweaked once you know what they are for. A complete reference to the UDP transport configuration can be found on the JBoss clustering guide ([http://docs.jboss.org/jbossclustering/cluster_guide/5.1/html/jgroups.chapt.html](http://docs.jboss.org/jbossclustering/cluster_guide/5.1/html/jgroups.chapt.html)); for the purpose of our book we will point out which are the most interesting ones for fine-tuning your transport. Here's the core section of the UDP transport configuration:

The biggest performance hit can be achieved by properly tuning the attributes concerning buffer size (`ucast_recv_buf_size, ucast_send_buf_size, mcast_recv_buf_size, and mcast_send_buf_size`).

For More Information:

<UDP
  singleton_name="shared-udp"
  mcast_port="@{jboss.jgroups.udp.mcast_port:45688}@"
  mcast_addr="@{jboss.partition.udpGroup:228.11.11.11}@"
  tos="8"
  ucast_recv_buf_size="20000000"
  ucast_send_buf_size="640000"
  mcast_recv_buf_size="25000000"
  mcast_send_buf_size="640000"
  loopback="true"
  discard_incompatible_packets="true"
  enable_bundling="false"
  max_bundle_size="64000"
  max_bundle_timeout="30"
...
/>

As a matter of fact, in order to guarantee optimal performance and adequate reliability of UDP multicast, it is essential to size network buffers correctly. Using inappropriate network buffers the chances are that you will experience a high frequency of UDP packets being dropped in the network layers, which therefore need to be retransmitted.

The default values for JGroups’ UDP transmission are 20MB and 64KB for unicast transmission and respectively 25MB and 64KB for multicast transmission. While these values sound appropriate for most cases, they can be insufficient for applications sending lots of cluster messages. Think about an application sending a thousand 1KB messages: with the default receive size, we will not be able to buffer all packets, thus increasing the chance of packet loss and costly retransmission.

For More Information:
Monitoring the intra-clustering traffic can be done through the `jboss.jgroups` domain Mbeans. For example, in order to monitor the amount of bytes sent and received with the UDP transmission protocol, just open your jmx-console and point at the `jboss.jgroups` domain. Then select your cluster partition. (Default the partition if you are running with default cluster settings). In the following snapshot (we are including only the relevant properties) we can see the amount of Messages sent/received along with their size (in bytes).

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Type</th>
<th>Description</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessagesSent</td>
<td>R</td>
<td>long</td>
<td>Mean Attribute</td>
<td>556</td>
</tr>
<tr>
<td>MessagesReceived</td>
<td>R</td>
<td>long</td>
<td>Mean Attribute</td>
<td>473</td>
</tr>
<tr>
<td>BytesSent</td>
<td>R</td>
<td>long</td>
<td>Mean Attribute</td>
<td>23740</td>
</tr>
<tr>
<td>BytesReceived</td>
<td>R</td>
<td>long</td>
<td>Mean Attribute</td>
<td>456066</td>
</tr>
</tbody>
</table>

Besides increasing the JGroups' buffer size, another important aspect to consider is that most operating systems allow a maximum UDP buffer size, which is generally lower than JGroups' defaults. For completeness, we include here a list of default maximum UDP buffer size:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Default Max UDP Buffer (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>131071</td>
</tr>
<tr>
<td>Windows</td>
<td>No known limit</td>
</tr>
<tr>
<td>Solaris</td>
<td>262144</td>
</tr>
<tr>
<td>FreeBSD, Darwin</td>
<td>262144</td>
</tr>
<tr>
<td>AIX</td>
<td>1048576</td>
</tr>
</tbody>
</table>

So, as a rule of thumb, you should always configure your operating system to take advantage of the JGroups' transport configuration. The following table shows the command required to increase the maximum buffer to 25 megabytes. You will need root privileges in order to modify these kernel parameters:

For More Information:
Another option that is worth trying is `enable_bundling`, which specifies whether to enable message bundling. If true, the transport protocol would queue outgoing messages until `max_bundle_size` bytes have accumulated, or `max_bundle_time` milliseconds have elapsed, whichever occurs first.

The advantage of using this approach is that the transport protocol would send bundled queued messages in one single larger message. Message bundling can have significant performance benefits for channels using asynchronous high volume messages (for example, JBoss Cache components configured for `REPL_ASYNC`.

On the other hand, for applications based on a synchronous exchange of RCPs, the introduction of message bundling would introduce a considerable latency so it is not recommended in this case. (That's the case with JBoss Cache components configured as `REPL_SYNC`).

### How to optimize the JGroups' Protocol stack

The Protocol stack contains a list of layers protocols, which need to be crossed by the message. A layer does not necessarily correspond to a transport protocol: for example a layer might take care to fragment the message or to assemble it. What's important to understand is that when a message is sent, it travels down in the stack, while when it's received it walks just the way back.

For example, in the next picture, the `FLUSH` protocol would be executed first, then the `STATE`, the `GMS`, and so on. Vice versa, when the message is received, it would meet the `PING` protocol first, then `MERGE2`, up to `FLUSH`.

---

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td><code>sysctl -w net.core.rmem_max=26214400</code></td>
</tr>
<tr>
<td>Solaris</td>
<td><code>ndd -set /dev/udp udp_max_buf 26214400</code></td>
</tr>
<tr>
<td>FreeBSD, Darwin</td>
<td><code>sysctl -w kern.ipc.maxsockbuf=26214400</code></td>
</tr>
<tr>
<td>AIX</td>
<td><code>no -o sb_max=8388608</code> (AIX will only allow 1MB, 4MB, or 8MB)</td>
</tr>
</tbody>
</table>

For More Information:
Following here, is the list of protocols triggered by the default UDP’s Protocol stack.

```xml
<stack name="udp"
     description="Default: IP multicast based stack, with flow control."/>
<config>
    <PING timeout="2000" num_initial_members="3"/>
    <MERGE2 max_interval="100000" min_interval="20000"/>
    <FD SOCK/>
    <FD timeout="6000" max_tries="5" shun="true"/>
    <VERIFY_SUSPECT timeout="1500"/>
    <pbcast.NAKACK use_mcast_xmit="false" gc_lag="0"
        retransmit_timeout="300,600,1200,2400,4800"
        discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200,2400,3600"/>
    <pbcast.STABLE stability_delay="1000"
        desired_avg_gossip="50000"
        max_bytes="400000"/>
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
        shun="true"
        view_bundling="true"
        view_ack_collection_timeout="5000"/>
    <FC max_credits="2000000" min_threshold="0.10"
        ignore_synchronous_response="true"/>
    <FRAG2 frag_size="60000"/>
    <pbcast.STATE_TRANSFER/>
    <pbcast.FLUSH timeout="0"/>
</config>
</stack>

The following table will shed some light on the above cryptic configuration:

<table>
<thead>
<tr>
<th>Category</th>
<th>Usage</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport</td>
<td>Responsible for sending and receiving messages across the network</td>
<td>IDP, TCP, and TUNNEL</td>
</tr>
<tr>
<td>Discovery</td>
<td>Used to discover active nodes in the cluster and determine which is the coordinator</td>
<td>PING, MPING, TCPPING, and TCPGOSSIP</td>
</tr>
<tr>
<td>Failure Detection</td>
<td>Used to poll cluster nodes to detect node failures</td>
<td>FD, FD_SIMPLE, FD_PING, FD_ICMP, FD.SOCK, and VERIFY_SUSPECT</td>
</tr>
</tbody>
</table>

For More Information:
### Category Usage Protocols

<table>
<thead>
<tr>
<th>Category</th>
<th>Usage</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable Delivery</td>
<td>Ensures that messages are actually delivered and delivered in the right order (FIFO) to the destination node</td>
<td>CAUSAL, NAKACK, pbcast. NAKACK, SMACK, UNICAST, and PBCAST</td>
</tr>
<tr>
<td>Group Membership</td>
<td>Used to notify the cluster when a node joins, leaves or crashes</td>
<td>pbcast.GMS, MERGE, MERGE2, and VIEW_SYNC</td>
</tr>
<tr>
<td>Flow Control</td>
<td>Used to adapt the data-sending rate to the data-receipt rate among nodes</td>
<td>FC</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>Fragments messages larger than a certain size. Unfragments at the receiver's side</td>
<td>FRAG2</td>
</tr>
<tr>
<td>State transfer</td>
<td>Synchronizes the application state (serialized as a byte array) from an existing node with a newly joining node</td>
<td>pbcast.STATE_TRANSFER and pbcast.STREAMING_STATE_TRANSFER</td>
</tr>
<tr>
<td>Distributed garbage collection</td>
<td>Periodically deletes those that have been seen by all nodes from the memory in each node</td>
<td>pbcast.STABLE</td>
</tr>
</tbody>
</table>

While all the above protocols play a role in message exchanging, it's not necessary that you know the inner details of all of them for tuning your applications. So we will focus just on a few interesting ones.

The FC protocol, for example can be used to adapt the rate of messages sent with the rate of messages received. This has the advantage of creating an homogeneous rate of exchange, where no sender member overwhelms receiver nodes, thus preventing potential problems like filling up buffers causing packet loss. Here's an example of FC configuration:

```xml
<FC max_credits="2000000"
    min_threshold="0.10"
    ignore_synchronous_response="true"/>
```

For More Information:  
The message rate adaptation is done with a simple credit system in which each time a sender sends a message a credit is subtracted (equal to the amount of bytes sent). Conversely, when a receiver collects a message, a credit is added.

- `max_credits` specifies the maximum number of credits (in bytes) and should obviously be smaller than the JVM heap size
- `min_threshold` specifies the value of `min_credits` as a percentage of the `max_credits` element
- `ignore_synchronous_response` specifies whether threads that have carried messages up to the application should be allowed to carry outgoing messages back down through FC without blocking for credits

The following image depicts a simple scenario where HostA is sending messages (and thus its `max_credits` is reduced) to HostB and HostC, which increase their `max_credits` accordingly.

![FC Protocol credit system](image)

The FC protocol, while providing a control over the flow of messages, can be a bad choice for applications that are issuing synchronous group RPC calls. In this kind of applications, if you have fast senders issuing messages, but some slow receivers across the cluster, the overall rate of calls will be slowed down. For this reason, remove FD from your protocol list if you are sending synchronous messages or just switch to the udp-sync protocol stack.

For More Information:
Besides JGroups, some network interface cards (NICs) and switches perform ethernet flow control (IEEE 802.3x), which causes overhead to senders when packet loss occurs. In order to avoid a redundant flow control, you are advised to remove ethernet flow control. For managed switches, you can usually achieve this via a web or Telnet/SSH interface. For unmanaged switches, unfortunately the only chance is to hope that ethernet flow control is disabled, or to replace the switch.

If you are using NICs, you can disable ethernet flow control by means of a simple shell command, for example, on Linux with the `ethtool`:

```
/sbin/ethtool -A eth0 autoneg off tx on rx on
```

If you want simply to verify if ethernet flow control is off:

```
/sbin/ethtool -a eth0
```

One more thing you must be aware of is that, by using JGroups, cluster nodes must store all messages received for potential retransmission in case of a failure. However, if we store all messages forever, we will run out of memory. The distributed garbage collection service in JGroups periodically removes messages that have been seen by all nodes from the memory in each node. The distributed garbage collection service is configured in the `pbcast.STABLE` sub-element like so:

```
<pbcast.STABLE stability_delay="1000"
    desired_avg_gossip="5000"
    max_bytes="400000"/>
```

The configurable attributes are as follows:

- `desired_avg_gossip`: Specifies the interval (in milliseconds) between garbage collection runs. Setting this parameter to 0 disables this service.
- `max_bytes`: Specifies the maximum number of bytes to receive before triggering a garbage collection run. Setting this parameter to 0 disables this service.

You are advised to set a `max_bytes` value if you have a high-traffic cluster.

For More Information:
Tuning JBoss Cache

JBoss Cache provides the foundation for many clustered services, which need to synchronize application state information across the set of nodes.

The cache is organized as a tree, with a single root. Each node in the tree essentially contains a map, which acts as a store for key/value pairs. The only requirement placed on objects that are cached is that they implement java.io.Serializable.

Actually EJB 3 Stateful Session Beans, HttpSession, and Entity/Hibernate rely on JBoss Cache to replicate information across the cluster. We have discussed thoroughly data persistence in Chapter 6, Tuning the Persistence Layer, so we will focus in the next sections on SFSB and HttpSession cluster tuning.

The core configuration of JBoss Cache is contained in the JBoss Cache Service. In JBoss AS 5, the scattered cache deployments have been replaced with a new CacheManager service, deployed via the <server>/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml.

The CacheManager acts as a factory for creating caches and as a registry for JBoss Cache instances. It is configured with a set of named JBoss Cache configurations. Here's a fragment of the standard SFSB cache configuration:

```xml
<entry>
  <key>sfsb-cache</key>
  <value>
    <bean name="StandardSFSBCacheConfig"
      class="org.jboss.cache.config.Configuration">
      <property name="clusterName">${jboss.partition.name:DefaultPartition}-SFSBCache</property>
      <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
      <property name="fetchInMemoryState">true</property>
      <property name="nodeLockingScheme">PESSIMISTIC</property>
      <property name="isolationLevel">REPEATABLE_READ</property>
    </bean>
  </value>
</entry>
```

For More Information:
Services that need a cache ask the CacheManager for the cache by name, which is specified by the key element; the cache manager creates the cache (if not already created) and returns it.

The simplest way to reference a custom cache is by means of the org.jboss.ejb3.annotation.CacheConfig annotation. For example, supposing you were to use a newly created Stateful Session Bean cache named custom_sfsb_cache:

```java
@Stateful
@Clustered
@CacheConfig(name="custom_sfsb_cache")
public Class SFSBExample {
}
```

The CacheManager keeps a reference to each cache it has created, so all services that request the same cache configuration name will share the same cache. When a service is done with the cache, it releases it to the CacheManager. The CacheManager keeps track of how many services are using each cache, and will stop and destroy the cache when all services have released it.

**Understanding JBoss Cache configuration**

In order to tune your JBoss Cache, it's essential to learn some key properties. In particular we need to understand:

- How data can be transmitted between its members. This is controlled by the cacheMode property.
- How the cache handles concurrency on data between cluster nodes. This is handled by nodeLockingScheme and isolationLevel configuration attributes.

**Configuring cacheMode**

The cacheMode property determines how JBoss Cache keeps in sync data across all nodes. Actually it can be split in two important aspects: how to notify changes across the cluster and how other nodes accommodate these changes on the local data.
As far data notification is concerned, there are the following choices:

- **Synchronous** means the cache instance sends a notification message to other nodes and before returning waits for them to acknowledge that they have applied the same changes. Waiting for acknowledgement from all nodes *adds delay*. However, if a synchronous replication returns successfully, the caller knows for sure that all modifications have been applied to all cache instances.

- **Asynchronous** means the cache instance sends a notification message and then immediately returns, without any acknowledgement that changes have been applied. The Asynchronous mode is most useful for cases like session replication (for example, Stateful Session Beans), where the cache sending data expects to be the only one that accesses the data. Asynchronous messaging adds a small potential risk that a fail over to another node may generate stale data, however, for many session-type applications this risk is acceptable given the major performance benefits gained.

- **Local** means the cache instance doesn't send a message at all. You should use this mode when you are running JBoss Cache as a single instance, so that it won't attempt to replicate anything. For example, JPA/Hibernate Query Cache uses a local cache to invalidate stale query result sets from the second level cache, so that JBoss Cache doesn't need to send messages around the cluster for a query result set cache.

As far as the second aspect is concerned (what should the other caches in the cluster do to reflect the change) you can distinguish between:

---

**For More Information:**

**Replication:** means that the cache replicates cached data across all cluster nodes. This means the sending node needs to include the changed state, increasing the cost of the message. Replication is necessary if the other nodes have no other way to obtain the state.

**Invalidation** means that you do not wish to replicate cached data but simply inform other caches in a cluster that data under specific addresses are now stale and should be evicted from memory. Invalidation reduces the cost of the cluster update messages, since only the cache key of the changed state needs to be transmitted, not the state itself.

By combining these two aspects we have a combination of five valid values for the `cacheMode` configuration attribute:

<table>
<thead>
<tr>
<th>CacheMode</th>
<th>Action</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPL_SYNC</td>
<td>Synchronous replication messages are sent</td>
<td>HAPartitionCache</td>
</tr>
<tr>
<td>REPL_ASYNC</td>
<td>Asynchronous replication messages are sent</td>
<td>EJB 3 SFSBs, HttpSession, Replicated QueryCache, TimestampCache</td>
</tr>
<tr>
<td>INVALIDATION_SYNC</td>
<td>Synchronous invalidation messages are sent</td>
<td>Entity and Collection Caches</td>
</tr>
<tr>
<td>INVALIDATION_ASYNC</td>
<td>Asynchronous invalidation messages are sent</td>
<td>None</td>
</tr>
<tr>
<td>LOCAL</td>
<td>No cluster messages are needed</td>
<td>Local Query cache</td>
</tr>
</tbody>
</table>
Should I use invalidation for session data?
No, you shouldn't. As a matter of fact, data invalidation it is an excellent option for a clustered JPA/Hibernate Entity cache, since the cached state can be re-read from the database in case of failure. If you use the invalidation option, with SFSBs or HttpSession, then you lose failover capabilities. If this matches with your project requirements, you could achieve better performance by simply turning off the cache.

Configuring cache concurrency
JBoss Cache is a thread-safe caching API, and uses its own efficient mechanisms of controlling concurrent access. Concurrency is configured via the nodeLockingScheme and isolationLevel configuration attributes.

There are three choices for nodeLockingScheme:

- **Pessimistic** locking involves threads/transactions acquiring locks on nodes before reading or writing. Which is acquired depends on the isolationLevel but in most cases a non-exclusive lock is acquired for a read and an exclusive lock is acquired for a write. Pessimistic locking requires a considerable overhead and allows lesser concurrency, since *reader threads must block until a write has completed* and released its exclusive lock (potentially a long time if the write is part of a transaction). The drawbacks include the potential for deadlocks, which are ultimately solved by a TimeoutException.

- **Optimistic** locking seeks to improve upon the concurrency available with Pessimistic by creating a workspace for each request/transaction that accesses the cache. All data is versioned; on completion of non-transactional requests or commits of transactions the version of data in the workspace is compared to the main cache, and an exception is raised if there are inconsistencies. This eliminates the cost of reader locks but, because of the cost associated with the parallel workspace, *it carries a high memory overhead and low scalability*.

- **MVCC** is the new locking schema that has been introduced in JBoss Cache 3.x (and packed with JBoss AS 5.x). In a nutshell, MVCC reduces the cost of slow, and synchronization-heavy schemas with a multi-versioned concurrency control, which is a locking scheme commonly used by modern database implementations to control concurrent access to shared data.

For More Information:
The most important features of MVCC are:

1. Readers don't acquire any locks.
2. Only one additional version is maintained for shared state, for a single writer.
3. All writes happen sequentially, to provide fail-fast semantics.

How does MVCC can achieve this?

For each reader thread, the MVCC's interceptors wraps state in a lightweight container object, which is placed in the thread's InvocationContext (or TransactionContext if running in a transaction). All subsequent operations on the state are carried out on the container object using Java references, which allow repeatable read semantics even if the actual state changes simultaneously.

Writer threads, on the other hand, need to acquire a lock before any writing can start. Currently, lock striping is used to improve the memory performance of the cache, and the size of the shared lock pool can be tuned using the concurrencyLevel attribute of the locking element.

After acquiring an exclusive lock on a cache Full Qualified Name, the writer thread then wraps the state to be modified in a container as well, just like with reader threads, and then copies this state for writing. When copying, a reference to the original version is still maintained in the container (for rollbacks). Changes are then made to the copy and the copy is finally written to the data structure when the write completes.

Should I use MVCC with session data too?

While MVCC is the default and recommended choice for JPA/Hibernate Entity caching, as far as Session caching is concerned, Pessimistic is still the default concurrency control. Why? As a matter of fact, accessing the same cached data by concurrent threads it's not the case with a user's session. This is strictly enforced in the case of SFSB, whose instances are not accessible concurrently. So don't bother trying to change this property for session data.

Configuring the isolationLevel

The isolationLevel attribute has two possible values, READ_COMMITTED and REPEATABLE_READ which correspond in semantics to database-style isolation levels. Previous versions of JBoss Cache supported all database isolation levels, and if an unsupported isolation level is configured, it is either upgraded or downgraded to the closest supported level.

For More Information:
REPEATABLE_READ is the default isolation level, to maintain compatibility with previous versions of JBoss Cache. READ_COMMITTED, while providing a slightly weaker isolation, has a significant performance benefit over REPEATABLE_READ.

**Tuning session replication**

As we have learnt, the user session needs replication in order to achieve a consistent state of your applications across the cluster. Replication can be a costly affair, especially if the amount of data held in session is significant. There are however some available strategies, which can mitigate a lot the cost of data replication and thus improve the performance of your cluster:

- **Override isModified method**: By including an `isModified` method in your SFSBs, you can achieve fine-grained control over data replication. Applicable to SFSBs only.
- **Use buddy replication**. By using buddy replication you are not replicating the session data to all nodes but to a limited set of nodes. Can be applicable both to SFSBs and HttpSession.
- **Configure replication granularity and replication trigger**. You can apply custom session policies to your HttpSession to define when data needs to be replicated and which elements need to be replicated as well. Applicable to HttpSession.

**Override SFSB’s isModified method**

One of the simplest ways to reduce the cost of SFSBs data replication is implementing in your EJB a method with the following signature: public boolean `isModified`();

Before replicating your bean, the container will detect if your bean implements this method. If your bean does, the container calls the `isModified` method and it *only replicates the bean when the method returns true.* If the bean has not been modified (or not enough to require replication, depending on your own preferences), you can return false and the replication will not occur.

If your session does not hold critical data (such as financial information), using the `isModified` method is a good option to achieve a substantial benefit in terms of performance. A good example could be a reporting application, which needs session management to generate aggregate reports through a set of wizards. Here's a graphical view of this process:
Chapter 7

The following benchmark is built on exactly the use case of an OLAP application, which uses SFSBs to drive some session data across a four step wizard. The benchmark compares the performance of the wizard without including `isModified` and by returning true to `isModified` at the end of the wizard.

Ultimately, by using the `isModified` method to propagate the session data at wider intervals you can improve the performance of your application with an acceptable risk to re-generate your reports in case of node failures.

Use buddy replication

By using buddy replication, sessions are replicated to a configurable number of backup servers in the cluster (also called buddies), rather than to all servers in the cluster. If a user fails over from the server that is hosting his or her session, the session data is transferred to the new server from one of the backup buddies. Buddy replication provides the following benefits:

- Reduced memory usage
- Reduced CPU utilization
- Reduced network transmission

For More Information:

The reason behind this large set of advantages is that each server only needs to store in its memory the sessions it is hosting as well as those of the servers for which it is acting as a backup. Thus, less memory required to store data, less CPU to elaborate bits to Java translations, and less data to transmit.

For example, in an 8-node cluster with each server configured to have one buddy, a server would just need to store 2 sessions instead of 8. That's just one fourth of the memory required with total replication.

In the following picture, you can see an example of a cluster configured for buddy replication:

Here, each node contains a cache of its session data and a backup of another node. For example, node A contains its session data and a backup of node E. Its data is in turn replicated to node B and so on.

In case of failure of node A, its data moves to node B which becomes the owner of both A and B data, plus the backup of node E. Node B in turn replicates (A + B) data to node C.
In order to configure your SFSB sessions or HttpSession to use buddy replication you have just to set to the property enabled of the bean BuddyReplicationConfig inside the <server>/deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml configuration file, as shown in the next code fragment:

```xml
<property name="buddyReplicationConfig">
    <bean class="org.jboss.cache.config.BuddyReplicationConfig">
        <property name="enabled">true</property>
        ...  
    </bean>
</property>
```

In the following test, we are comparing the throughput of a 5-node clustered web application which uses buddy replication against one which replicates data across all members of the cluster.

<table>
<thead>
<tr>
<th>Label</th>
<th># Samples</th>
<th>Average</th>
<th>Median</th>
<th>90% Line</th>
<th>Min</th>
<th>Max</th>
<th>Error %</th>
<th>Throughput</th>
<th>KB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Req.</td>
<td>150960</td>
<td>187</td>
<td>186</td>
<td>236</td>
<td>183</td>
<td>1439</td>
<td>0.09%</td>
<td>55.94MB/s</td>
<td>51.4</td>
</tr>
<tr>
<td>TOTAL</td>
<td>150960</td>
<td>187</td>
<td>186</td>
<td>236</td>
<td>183</td>
<td>1439</td>
<td>0.09%</td>
<td>55.94MB/s</td>
<td>51.4</td>
</tr>
</tbody>
</table>

**Web application benchmark using buddy replication**

<table>
<thead>
<tr>
<th>Label</th>
<th># Samples</th>
<th>Average</th>
<th>Median</th>
<th>90% Line</th>
<th>Min</th>
<th>Max</th>
<th>Error %</th>
<th>Throughput</th>
<th>KB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Req.</td>
<td>150960</td>
<td>200</td>
<td>176</td>
<td>251</td>
<td>187</td>
<td>1551</td>
<td>0.09%</td>
<td>50.95MB/s</td>
<td>41.2</td>
</tr>
<tr>
<td>TOTAL</td>
<td>150960</td>
<td>200</td>
<td>176</td>
<td>251</td>
<td>187</td>
<td>1551</td>
<td>0.09%</td>
<td>50.95MB/s</td>
<td>41.2</td>
</tr>
</tbody>
</table>

**Web application benchmark with full cluster replication**

For More Information:  
In this benchmark, switching on buddy replication improved the application throughput of about 30%. No doubt that by using buddy replication there's a high potential for scaling because memory/CPU/network usage per node does not increase linearly as new nodes are added.

**Advanced buddy replication**

With the minimal configuration we have just described, each server will look for one buddy across the network where data needs to be replicated. If you need to backup your session to a larger set of buddies you can modify the `numBuddies` property of the `BuddyReplicationConfig` bean. Consider, however, that replicating the session to a large set of nodes would conversely reduce the benefits of buddy replication.

Still using the default configuration, each node will try to select its buddy on a different physical host: this helps to reduce chances of introducing a single point of failure in your cluster. Just in case the cluster node is not able to find buddies on different physical hosts, it will not honour the property `ignoreColocatedBuddies` and fall back to co-located nodes.

The default policy is often what you might need in your applications, however if you need a fine-grained control over the composition of your buddies you can use a feature named *buddy pool*. A buddy pool is an optional construct where each instance in a cluster may be configured to be part of a group- just like an "exclusive club membership".

This allows system administrators a degree of flexibility and control over how buddies are selected. For example, you might put two instances on separate physical servers that may be on two separate physical racks in the same buddy pool. So rather than picking an instance on a different host on the same rack, the `BuddyLocators` would rather pick the instance in the same buddy pool, on a separate rack which may add a degree of redundancy.

Here's a complete configuration which includes buddy pools:

```xml
<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">
    <property name="enabled">true</property>
    <property name="buddyPoolName">rack1</property>
    <property name="buddyCommunicationTimeout">17500</property>
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>
  </bean>
</property>
```

For More Information:

<property name="buddyLocatorConfig">
  <bean class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
    <property name="numBuddies">1</property>
    <property name="ignoreColocatedBuddies">true</property>
  </bean>
</property>

In this configuration fragment, the buddyPoolName element, if specified, creates a logical subgroup and only picks buddies who share the same buddy pool name. If not specified, this defaults to an internal constant name, which then treats the entire cluster as a single buddy pool.

If the cache on another node needs data that it doesn't have locally, it can ask the other nodes in the cluster to provide it; nodes that have a copy will provide it as part of a process called data gravitation. The new node will become the owner of the data, placing a backup copy of the data on its buddies.

The ability to gravitate data means there is no need for all requests for data to occur on a node that has a copy of it; that is, any node can handle a request for any data. However, data gravitation is expensive and should not be a frequent occurrence; ideally it should only occur if the node that is using some data fails or is shut down, forcing interested clients to fail over to a different node.

The following optional properties pertain to data gravitation:

- **autoDataGravitation**: Whether data gravitation occurs for every cache miss. By default this is set to false to prevent unnecessary network calls.
- **DataGravitationRemoveOnFind**: Forces all remote caches that own the data or hold backups for the data to remove that data, thereby making the requesting cache the new data owner. If set to false, an evict is broadcast instead of a remove, so any state persisted in cache loaders will remain. This is useful if you have a shared cache loader configured. (See next section about Cache loader). Defaults to true.
- **dataGravitationSearchBackupTrees**: Asks remote instances to search through their backups as well as main data trees. Defaults to true. The resulting effect is that if this is true then backup nodes can respond to data gravitation requests in addition to data owners.

For More Information:
**Buddy replication and session affinity**

One of the pre-requisites to buddy replication working well and being a real benefit is the use of session affinity, also known as **sticky sessions** in HttpSession replication speak. What this means is that if certain data is frequently accessed, it is desirable that this is always accessed on one instance rather than in a "round-robin" fashion as this helps the cache cluster optimise how it chooses buddies, where it stores data, and minimises replication traffic.

If you are replicating SFSBs session, there is no need to configure anything since SFSBs, once created, are pinned to the server that created them.

When using HttpSession, you need to make sure your software or hardware load balancer maintain the session on the same host where it was created.

By using Apache's mod_jk, you have to configure the workers file (workers.properties) specifying where the different node and how calls should be load-balanced across them. For example, on a 5-node cluster:

```ini
worker.loadbalancer.balance_workers=node1,node2,node3,node4,node5
worker.loadbalancer.sticky_session=1
```

Basically, the above snippet configures mod_jk to perform round-robin load balancing with sticky sessions (sticky_session=1) across 5 nodes of a cluster.

**Configure replication granularity and replication trigger**

Applications that want to store data in the HttpSession need to use the methods `setAttribute` to store the attributes and `getAttribute` to retrieve them. You can define two kind of properties related to HttpSession:

- The **replication-trigger** configures when data needs to be replicated.
- The **replication-granularity** defines which part of the session needs to be replicated.

Let's dissect both aspects in the following sections:
How to configure the replication-trigger

The replication-trigger element determines what triggers a session replication and can be configured by means of the jboss-web.xml element (packed in the WEB-INF folder of your web application). Here’s an example:

```
<jboss-web>
  <replication-config>
    <replication-trigger>SET</replication-trigger>
  </replication-config>
</jboss-web>
```

The following is a list of possible alternative options:

- **SET_AND_GET** is conservative but not performance-wise; it will always replicate session data even if its content has not been modified but simply accessed. This setting made (a little) sense in AS 4 since using it was a way to ensure that every request triggered replication of the session’s timestamp. Setting max_unreplicated_interval to 0 accomplishes the same thing at much lower cost.

- **SET_AND_NON_PRIMITIVE_GET** is conservative but will only replicate if an object of a non-primitive type has been accessed (that is, the object is not of a well-known immutable JDK type such as Integer, Long, String, and so on.) This is the default value.

- **SET** assumes that the developer will explicitly call setAttribute on the session if the data needs to be replicated. This setting prevents unnecessary replication and can have a major beneficial impact on performance.

In all cases, calling setAttribute marks the session as dirty and thus triggers replication.

For More Information:

JBoss AS Cluster Tuning

For the purpose of evaluating the available alternatives in performance terms, we have compared a benchmark of a web application using different replication-triggers:

- In the first benchmark, we are using the default rule (SET_AND_NON_PRIMITIVE_GET).
- In the second we have switched to SET policy, issuing a setAttribute on 50% of the requests.
- In the last benchmark, we have formerly populated the session with the required attributes and then issued only queries on the session via the getAttribute method.

As you can see the benefit of using the SET replication trigger is obvious, especially if you follow a read-mostly approach on non-primitive types. On the other hand, this requires very good coding practices to ensure setAttribute is always called whenever a mutable object stored in the session is modified.

How to configure the replication-granularity

As far as what data needs to be replicated is concerned, you can opt for the following choices:

- **SESSION** indicates that the *entire session* attribute map should be replicated when any attribute is considered modified. Replication occurs at request end. This option replicates the most data and thus incurs the highest replication cost, but since all attributes values are always replicated together it ensures that any references between attribute values will not be broken when the session is deserialized. For this reason it is the default setting.

- **ATTRIBUTE** indicates that *only attributes* that the session considers to be potentially modified are replicated. Replication occurs at request end. For sessions carrying large amounts of data, parts of which are infrequently updated, this option can significantly increase replication performance.

For More Information:

• **FIELD** level replication only replicates modified data fields inside objects stored in the session. Its use could potentially drastically reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use **FIELD**-level replication, you have to first prepare (that is bytecode enhance) your Java class to allow the session cache to detect when fields in cached objects have been changed and need to be replicated.

In order to change the default replication granularity, you have to configure the desired attribute in your `jboss-web.xml` configuration file:

```
<jboss-web>
  <replication-config>
    <replication-granularity>FIELD</replication-granularity>
    <replication-field-batch-mode>true</replication-field-batch-mode>
  </replication-config>
</jboss-web>
```

In the above example, the replication-field-batch-mode element indicates whether you want all replication messages associated with a request to be batched into one message.

Additionally, if you want to use **FIELD** level replication you need to perform a bit of extra work. At first you need to add the `@org.jboss.cache.pojo.annotation.Replicable` annotation at class level:

```java
@Replicable
public class Person { ... }
```

For More Information:
If you annotate a class with @Replicable, then all of its subclasses will be automatically annotated as well.

Once you have annotated your classes, you will need to perform a post-compiler processing step to bytecode enhance your classes for use by your cache. Please check the JBoss AOP documentation (http://www.jboss.org/jbossaop) for the usage of the aoc post-compiler. The JBoss AOP project also provides easy to use ANT tasks to help integrate those steps into your application build process.

As proof of concept, let's build a use case to compare the performance of ATTRIBUTE and FIELD granularity policies. Supposing you are storing in your HttpSession an object of Person type. The object contains references to an Address, ContactInfo, and PersonalInfo objects. It contains also an ArrayList of WorkExperience.

A prerequisite to this benchmark is that there are no references between the field values stored in the Person class (for example between the contactInfo and personalInfo fields), otherwise the references will be broken by ATTRIBUTE or FIELD policies.

By using the SESSION or ATTRIBUTE replication-granularity policy, even if just one of these fields is modified, the whole Person object need to be retransmitted. Let's compare the throughput of two applications using respectively the ATTRIBUTE and FIELD replication-granularity.

For More Information:
In this example, based on the assumption that we have a single dirty field of Person' class per request, by using FIELD Replication generate a substantial 10% gain.

**Tuning cache storage**

Cache loading allows JBoss Cache to store cached data in a persistent store and is used mainly for HttpSession and SFSB sessions. Hibernate and JPA on the other hand, have already their persistence storage in the database so it doesn't make sense to add another storage.

This data can either be an overflow, where the data in the persistent store has been evicted from memory. Or it can be a replication of what is in memory, where everything in memory is also reflected in the persistent store, along with items that have been evicted from memory.

The cache storage used for web session and EJB3 SFSB caching comes into play in two circumstances:

- Whenever a cache element is accessed, and that element is not in the cache (for example, due to eviction or due to server restart), then the cache loader transparently loads the element into the cache if found in the backend store.
- Whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader (except if the ignoreModifications property has been set to true for a specific cache loader). If transactions are used, all modifications created within a transaction are persisted as well.

Cache loaders are configured by means of the property cacheLoaderConfig of session caches. For example, in the case of SFSB cache:

```xml
<entry><key>sfsb-cache</key>
  <value>
    <bean name="StandardSFSBCacheConfig" class="org.jboss.cache.config.Configuration">
      ...
      <property name="cacheLoaderConfig">
```

For More Information:
The passivation property, when set to true, means the persistent store acts as an overflow area written to when data is evicted from the in-memory cache.

The shared attribute indicates that the cache loader is shared among different cache instances, for example where all instances in a cluster use the same JDBC settings to talk to the same remote, shared database. Setting this to true prevents repeated and unnecessary writes of the same data to the cache loader by different cache instances. The default value is false.

**Where does cache data get stored?**

By default, the Cache loader uses a filesystem implementation based on the class org.jboss.cache.loader.FileCacheLoaderConfig, which requires the location property to define the root directory to be used.

If set to true, the async attribute read operations are done synchronously, while write (CRUD - Create, Remove, Update, and Delete) operations are done asynchronously. If set to false (default), both read and writes are performed synchronously.
Should I use an async channel for my Cache Loader?

When using an async channel, an instance of `org.jboss.cache.loader.AsyncCacheLoader` is constructed which will act as an asynchronous channel to the actual cache loader to be used. Be aware that, using the `AsyncCacheLoader`, there is always the possibility of dirty reads since all writes are performed asynchronously, and it is thus impossible to guarantee when (and even if) a write succeeds. On the other hand the `AsyncCacheLoader` allows massive writes to be written asynchronously, possibly in batches, with large performance benefits. Checkout the JBoss Cache docs for further information [http://docs.jboss.org/jbosscache/3.2.1.GA/apidocs/index.html](http://docs.jboss.org/jbosscache/3.2.1.GA/apidocs/index.html).

`fetchPersistentState` determines whether or not to fetch the persistent state of a cache when a node joins a cluster and conversely the `purgeOnStartup` property evicts data from the storage on startup, if set to true.

Finally, `checkCharacterPortability` should be false for a minor performance improvement.

The `FileCacheLoader` is a good choice in terms of performance, however it has some limitations, which you should be aware of before rolling your application in a production environment. In particular:

1. Due to the way the `FileCacheLoader` represents a tree structure on disk (directories and files) traversal is "inefficient" for deep trees.
2. Usage on shared filesystems such as NFS, Windows shares, and others should be avoided as these do not implement proper file locking and can cause data corruption.
3. Filesystems are inherently not "transactional", so when attempting to use your cache in a transactional context, failures when writing to the file (which happens during the commit phase) cannot be recovered.

As a rule of thumb, it is recommended that the `FileCacheLoader` not be used in a highly concurrent, transactional, or stressful environment, and, in this kind of scenario consider using it just in the testing environment.

For More Information:
As an alternative, consider that JBoss Cache is distributed with a set of different Cache loaders which can be used as alternative. For example:

- The JDBC-based cache loader implementation that stores/loads nodes' state into a relational database. The implementing class is `org.jboss.cache.loader.JDBCCacheLoader`.

- The `BdbjeCacheLoader`, which is a cache loader implementation based on the Oracle/Sleepycat's BerkeleyDB Java Edition (note that the BerkeleyDB implementation is much more efficient than the filesystem-based implementation, and provides transactional guarantees, but requires a commercial license if distributed with an application (see [http://www.oracle.com/database/berkeley-db/index.html](http://www.oracle.com/database/berkeley-db/index.html) for details).

- The `JdbmCacheLoader`, which is a cache loader implementation based on the JDBM engine, a fast and free alternative to BerkeleyDB.

- Finally, `S3CacheLoader`, which uses the Amazon S3 solution (Simple Storage Solution [http://aws.amazon.com/](http://aws.amazon.com/)) for storing cache data. Since Amazon S3 is remote network storage and has fairly high latency, it is really best for caches that store large pieces of data, such as media or files.

When it comes to measuring the performance of different Cache Loaders, here's a benchmark executed to compare the File CacheLoader, the JDBC CacheLoader (based on Oracle Database) and Jdbm CacheLoader.

![Graph showing performance comparison between File CacheLoader, JDBC CacheLoader, and Jdbm CacheLoader](image)

In the above benchmark we are testing cache insertion and cache gets of batches of 1000 Fqn each one bearing 10 attributes. The File CacheLoader accomplished the overall best performance, while the JBDM CacheLoader is almost as fast for Cache gets.

The JDBC CacheLoader is the most robust solution but it adds more overhead to the Cache storage of your session data.
Summary
Clustering is a key element in building scalable Enterprise applications. The infrastructure used by JBoss AS for clustered applications is based on JGroups framework for the nodes inter-communication and JBoss Cache for keeping the cluster data synchronized across nodes.

- **JGroups** can use both UDP and TCP as communication protocol. Unless you have network restriction, you should stay with the default UDP that uses multicast to send and receive messages.
  - You can tune the transmission protocol by setting an appropriate buffer size with the properties `mcast_recv_buf_size`, `mcast_send_buf_size`, `ucast_recv_buf_size`, and `ucast_send_buf_size`. You should as well increase your O/S buffer size, which need to be adequate to accept JGroups' settings.
- **JBoss Cache** provides the foundation for robust clustered services.
- By configuring the `cacheMode` you can choose if your cluster messages will be synchronous (that is will wait for message acknowledgement) or asynchronous. Unless you need to handle cache message exceptions, stay with the asynchronous pattern, which provides the best performance.
- Cache messages can trigger as well cluster replication or cluster invalidation. A cluster replication is needed for transferring the session state across the cluster while invalidation is the default for Entity/Hibernate, where state can be recovered from the database.
- The cache concurrency can be configured by means of the `nodeLockingScheme` property. The most efficient locking schema is the MVCC, which reduces the cost of slow, or synchronization-heavy schemas of Pessimistic and Optimistic schemas.
- Cache replication of sessions can be optimised mostly in three ways:
  - By overriding the `isModified` method of your SFSBs you can achieve a fine-grained control over data replication. It's an optimal quick-tuning option for OLAP applications using SFSBs.
  - **Buddy replication** is the most important performance addition to your session replication. It helps to increase the performance by reducing memory and CPU usage as well as network traffic. Use buddy replication pools to achieve a higher level of redundancy for mission critical applications.

For More Information:
Clustered web applications can configure **replication-granularity** and **replication-trigger**:

- As far as the replication trigger is concerned, if you mostly read immutable data from your session, the `SET` attribute provides a substantial benefit over the default `SET_AND_PRIMITIVE_GET`.

- As far as replication granularity is concerned, if your sessions are generally small, you can stay with the default policy (SESSION). If your session is larger and some parts are infrequently accessed, `ATTRIBUTE` replication will be more effective. If your application has very big data objects in session attributes and only fields in those objects are frequently modified, the `FIELD` policy would be the best.

For More Information:

Where to buy this book
You can buy JBoss AS 5 Performance Tuning from the Packt Publishing website:
Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.