JBoss AS 5.1 Clustering Guide

High Availability Enterprise Services with JBoss Application Server Clusters

by Brian Stansberry, Galder Zamarreno, and Paul Ferraro

edited by Samson Kittoli

Part I. Introduction and Core Services

Introduction and Quick Start

Clustering allows you to run an application on several parallel servers (a.k.a cluster nodes) while providing a single view to application clients. Load is distributed across different servers, and even if one or more of the servers fails, the application is still accessible via the surviving cluster nodes. Clustering is crucial for scalable enterprise applications, as you can improve performance by adding more nodes to the cluster. Clustering is crucial for highly available enterprise applications, as it is the clustering infrastructure that supports the redundancy needed for high availability.

The JBoss Application Server (AS) comes with clustering support out of the box, as part of the `all` configuration. The `all` configuration includes support for the following:

- A scalable, fault-tolerant JNDI implementation (HA-JNDI).
- Web tier clustering, including:
  - High availability for web session state via state replication.
  - Ability to integrate with hardware and software load balancers, including special integration with mod_jk and other JK-based software load balancers.
  - Single Sign-on support across a cluster.
- EJB session bean clustering, for both stateful and stateless beans, and for both EJB3 and EJB2.
- A distributed cache for JPA/Hibernate entities.
- A framework for keeping local EJB2 entity caches consistent across a cluster by invalidating cache entries across the cluster when a bean is changed on any node.
- Distributed JMS queues and topics via JBoss Messaging.
- Deploying a service or application on multiple nodes in the cluster but having it active on only one (but at least one) node, a.k.a. an "HA Singleton".

In this *Clustering Guide* we aim to provide you with an in depth understanding of how to use JBoss AS's clustering features. In this first part of the guide, the goal is to provide some basic "Quick Start" steps to encourage you to start experimenting with JBoss AS Clustering, and then to provide some background information that will allow you to understand how JBoss AS Clustering works. The next part of the guide then explains in detail how to use these features to cluster your JEE services. Finally, we provide some more details about advanced configuration of JGroups and JBoss Cache, the core technologies that underlie JBoss AS Clustering.

1.1. Quick Start Guide

The goal of this section is to give you the minimum information needed to let you get started experimenting with JBoss AS Clustering. Most of the areas touched on in this section are covered in much greater detail later in this guide.

1.1.1. Initial Preparation

Preparing a set of servers to act as a JBoss AS cluster involves a few simple steps:

- **Install JBoss AS on all your servers.** In its simplest form, this is just a matter of unzipping the JBoss download onto the filesystem on each server.

  If you want to run multiple JBoss AS instances on a single server, you can either install the full JBoss distribution onto multiple locations on your filesystem, or you can simply make copies of the `all` configuration. For example, assuming the root of the JBoss distribution was unzipped to `/var/jboss`, you would:

```
$ cd /var/jboss/server
$ cp -r all node1
$ cp -r all node2
```

- **For each node, determine the address to bind sockets to.** When you start JBoss, whether clustered or not, you need to tell JBoss on what address its sockets should listen for traffic. (The default is `localhost` which is secure but isn't very useful, particularly in a cluster.) So, you need to decide what those addresses will be.

- **Ensure multicast is working.** By default JBoss AS uses UDP multicast for most intra-cluster communications. Make sure each server's networking configuration supports multicast and that multicast support is enabled for any switches or routers between your servers. If you are planning to run more than one node on a server, make sure the server's routing table includes a multicast route. See the JGroups documentation at *http://www.jgroups.org* for more on this general area, including information on how to use JGroups' diagnostic tools to confirm that multicast is working.

  > **Note**
  >
  > JBoss AS clustering does not require the use of UDP multicast; the AS can also be reconfigured to use TCP unicast for intra-cluster communication.

- **Determine a unique integer "ServerPeerID" for each node.** This is needed for JBoss Messaging clustering, and can be skipped if you will not be running JBoss Messaging (i.e. you will remove JBM from your server configuration's `deploy` directory). JBM requires that each node in a cluster has a unique integer id, known as a "ServerPeerID", that should remain consistent across server restarts. A simple 1, 2, 3, ..., x naming scheme is fine. We'll cover how to use these integer ids in the next section.

Beyond the above required steps, the following two optional steps are recommended to help ensure that your cluster is properly isolated from other JBoss AS clusters that may be running on your network:

- **Pick a unique name for your cluster.** The default name for a JBoss AS cluster is "DefaultPartition". Come up with a different name for each cluster in your environment, e.g. "QAPartition" or "BobsDevPartition". The use of "Partition" is not required; it's just a semi-convention. As a small aid to performance try to keep the name short, as it gets included in every message sent around the cluster. We'll cover how to use the name you pick in the next section.
- **Pick a unique multicast address for your cluster.** By default JBoss AS uses UDP multicast for most intra-cluster communication. Pick a different multicast address for each cluster you run. Generally a good multicast address is of the form `239.255.x.y`. See *http://www.29west.com/docs/THPM/multicast-address-assignment.html* [http://www.29west.com/docs/THPM/multicast-address-assignment.html] for a good discussion on multicast address assignment. We'll cover how to use the address you pick in the next section.

See *Section 10.2.2, "Isolating JGroups Channels"* for more on isolating clusters.

1.1.2. Launching a JBoss AS Cluster

The simplest way to start a JBoss server cluster is to start several JBoss instances on the same local network, using the `-c all` command line option for each instance. Those server instances will detect each other and automatically form a cluster.

Let's look at a few different scenarios for doing this. In each scenario we'll be creating a two node cluster, where the *ServerPeerID [4]* for the first node is `1` and for the second node is `2`. We've decided to call our cluster "DocsPartition" and to use `239.255.100.100` as our multicast address. These scenarios are meant to be illustrative; the use of a two node cluster shouldn't be taken to mean that is the best size for a cluster; it's just that's the simplest way to do the examples.

- **Scenario 1: Nodes on Separate Machines**

  This is the most common production scenario. Assume the machines are named "node1" and "node2", while node1 has an IP address of `192.168.0.101` and node2 has an address of `192.168.0.102`. Assume the "ServerPeerID" for node1 is `1` and for node2 it's `2`. Assume on each machine JBoss is installed in `/var/jboss`.

  On node1, to launch JBoss:

```
$ cd /var/jboss/bin
$ ./run.sh -c all -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1
```

  On node2, it's the same except for a different `-b` value and ServerPeerID:

```
$ cd /var/jboss/bin
$ ./run.sh -c all -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.102 -Djboss.messaging.ServerPeerID=2
```

The `-c` switch says to use the `all` config, which includes clustering support. The `-g` switch sets the cluster name. The `-u` switch sets the multicast address that will be used for intra-cluster communication. The `-b` switch sets the address on which sockets will be bound. The `-D` switch sets system property `jboss.messaging.ServerPeerId`, from which JBoss Messaging gets its unique id.

- **Scenario 2: Two Nodes on a Single, Multihomed, Server**

  Running multiple nodes on the same machine is a common scenario in a development environment, and is also used in production in combination with Scenario 1. (Running *all* the nodes in a production cluster on a single machine is generally not recommended, since the machine itself becomes a single point of failure.) In this version of the scenario, the machine is multihomed, i.e. has more than one IP address. This allows the binding of each JBoss instance to a different address, preventing port conflicts when the nodes open sockets.

  Assume the single machine has the `192.168.0.101` and `192.168.0.102` addresses assigned, and that the two JBoss instances use the same addresses and ServerPeerIDs as in Scenario 1. The difference from Scenario 1 is we need to be sure each AS instance has its own work area. So, instead of using the `all` config, we are going to use the `node1` and `node2` configs we copied from `all` in *the previous section [4]*.

  To launch the first instance, open a console window and:

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1
```

  For the second instance, it's the same except for different *-b* and *-c* values and a different ServerPeerID:

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.102 -Djboss.messaging.ServerPeerID=2
```

- **Scenario 3: Two Nodes on a Single, Non-Multihomed, Server**

  This is similar to Scenario 2, but here the machine only has one IP address available. Two processes can't bind sockets to the same address and port, so we'll have to tell JBoss to use different ports for the two instances. This can be done by configuring the ServiceBindingManager service by setting the `jboss.service.binding.set` system property.

  To launch the first instance, open a console window and:

```
$ cd /var/jboss/bin
$ ./run.sh -c node1 -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.101 -Djboss.messaging.ServerPeerID=1 \
   -Djboss.service.binding.set=ports-default
```

  For the second instance:

```
$ cd /var/jboss/bin
$ ./run.sh -c node2 -g DocsPartition -u 239.255.100.100 \
   -b 192.168.0.101 -Djboss.messaging.ServerPeerID=2 \
   -Djboss.service.binding.set=ports-01
```

  This tells the ServiceBindingManager on the first node to use the standard set of ports (e.g. JNDI on 1099). The second node uses the "ports-01" binding set, which by default for each port has an offset of 100 from the standard port number (e.g. JNDI on 1199). See the `conf/bootstrap/bindings.xml` file for the full ServiceBindingManager configuration.

  Note that this setup is not advised for production use, due to the increased management complexity that comes with using different ports. But it is a fairly common scenario in development environments where developers want to use clustering but cannot multihome their workstations.

> **Note**
>
> Including `-Djboss.service.binding.set=ports-default` on the command line for node1 isn't technically necessary, since `ports-default` is the ... default. But using a consistent set of command line arguments across all servers is helpful to people less familiar with all the details.

That's it; that's all it takes to get a cluster of JBoss AS servers up and running.

1.1.3. Web Application Clustering Quick Start

JBoss AS supports clustered web sessions, where a backup copy of each user's `HttpSession` state is stored on one or more nodes in the cluster. In case the primary node handling the session fails or is shut down, any other node in the cluster can handle subsequent requests for the session by accessing the backup copy. Web tier clustering is discussed in detail in *Chapter 8, HTTP Services*.

There are two aspects to setting up web tier clustering:

- **Configuring an External Load Balancer**. Web applications require an external load balancer to balance HTTP requests across the cluster of JBoss AS instances (see *Section 2.2.2, "External Load Balancer Architecture"* for more on why that is). JBoss AS itself doesn't act as an HTTP load balancer. So, you will need to set up a hardware or software load balancer. There are many possible load balancer choices, so how to configure one is really beyond the scope of a Quick Start. But see *Section 8.1, "Configuring load balancing using Apache and mod_jk"* for details on how to set up the popular mod_jk software load balancer.
- **Configuring Your Web Application for Clustering**. This aspect involves telling JBoss you want clustering behavior for a particular web app, and it couldn't be simpler. Just add an empty `distributable` element to your application's `web.xml` file:

```
<?xml version="1.0"?>
<web-app  xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        version="2.5">

   <distributable/>

</web-app>
```

  Simply doing that is enough to get the default JBoss AS web session clustering behavior, which is appropriate for most applications. See *Section 8.2, "Configuring HTTP session state replication"* for more advanced configuration options.

1.1.4. EJB Session Bean Clustering Quick Start

JBoss AS supports clustered EJB session beans, whereby requests for a bean are balanced across the cluster. For stateful beans a backup copy of bean state is maintained on one or more cluster nodes, providing high availability in case the node handling a particular session fails or is shut down. Clustering of both EJB2 and EJB3 beans is supported.

For EJB3 session beans, simply add the `org.jboss.ejb3.annotation.Clustered` annotation to the bean class for your stateful or stateless bean:

```
@javax.ejb.Stateless
@org.jboss.ejb3.annotation.Clustered
public class MyBean implements MySessionInt {

  public void test() {
    // Do something cool
  }
}
```

For EJB2 session beans, or for EJB3 beans where you prefer XML configuration over annotations, simply add a `clustered` element to the bean's section in the JBoss-specific deployment descriptor, `jboss.xml`:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>example.StatelessSession</ejb-name>
      <jndi-name>example.StatelessSession</jndi-name>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>
</jboss>
```

See *Chapter 6, Clustered Session EJBs* for more advanced configuration options.

1.1.5. Entity Clustering Quick Start

One of the big improvements in the clustering area in JBoss AS 5 is the use of the new Hibernate/JBoss Cache integration for second level entity caching that was introduced in Hibernate 3.3. In the JPA/Hibernate context, a second level cache refers to a cache whose contents are retained beyond the scope of a transaction. A second level cache *may* improve performance by reducing the number of database reads. You should always load test your application with second level caching enabled and disabled to see whether it has a beneficial impact on your particular application.

If you use more than one JBoss AS instance to run your JPA/Hibernate application and you use second level caching, you must use a cluster-aware cache. Otherwise a cache on server A will still hold out-of-date data after activity on server B updates some entities.

JBoss AS provides a cluster-aware second level cache based on JBoss Cache. To tell JBoss AS's standard Hibernate-based JPA provider to enable second level caching with JBoss Cache, configure your `persistence.xml` as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="somename" transaction-type="JTA">
    <jta-data-source>java:/SomeDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache" value="true"/>
      <property name="hibernate.cache.region.factory_class"
              value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
     <property name="hibernate.cache.region.jbc2.cachefactory" value="java:CacheManager"/>
      <!-- Other configuration options ... -->
    </properties>
  </persistence-unit>
</persistence>
```

That tells Hibernate to use the JBoss Cache-based second level cache, but it doesn't tell it what entities to cache. That can be done by adding the `org.hibernate.annotations.Cache` annotation to your entity class:

```
package org.example.entities;

import java.io.Serializable;
import javax.persistence.Entity;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cache (usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable
```

See *Chapter 7, Clustered Entity EJBs* for more advanced configuration options and details on how to configure the same thing for a non-JPA Hibernate application.

Note

Clustering can add significant overhead to a JPA/Hibernate second level cache, so don't assume that just because second level caching adds a benefit to a non-clustered application that it will be beneficial to a clustered application. Even if clustered second level caching is beneficial overall, caching of more frequently modified entity types may be beneficial in a non-clustered scenario but not in a clustered one. *Always* load test your application.

Clustering Concepts

In the next section, we discuss basic concepts behind JBoss' clustering services. It is helpful that you understand these concepts before reading the rest of the *Clustering Guide*.

2.1. Cluster Definition

A cluster is a set of nodes that communicate with each other and work toward a common goal. In a JBoss Application Server cluster (also known as a “partition”), a node is an JBoss Application Server instance. Communication between the nodes is handled by the JGroups group communication library, with a JGroups `Channel` providing the core functionality of tracking who is in the cluster and reliably exchanging messages between the cluster members. JGroups channels with the same configuration and name have the ability to dynamically discover each other and form a group. This is why simply executing “run -c all” on two AS instances on the same network is enough for them to form a cluster – each AS starts a `Channel` (actually, several) with the same default configuration, so they dynamically discover each other and form a cluster. Nodes can be dynamically added to or removed from clusters at any time, simply by starting or stopping a `Channel` with a configuration and name that matches the other cluster members.

On the same AS instance, different services can create their own `Channel`, and sometimes more than one. In a standard startup of the AS 5 *all* configuration, two different services create a total of four different channels – JBoss Messaging creates two and a core general purpose clustering service known as HAPartition creates two more. If you deploy clustered web applications, clustered EJB3 SFSBs or a clustered JPA/Hibernate entity cache, additional channels will be created. The channels the AS connects can be divided into three broad categories: a general purpose channel used by the HAPartition service, channels created by JBoss Cache for special purpose caching and cluster wide state replication, and two channels used by JBoss Messaging.

So, if you go to two AS 5.x instances and execute `run -c all`, the channels created on each server will discover each other and you'll have a conceptual cluster. It's easy to think of this as a two node cluster, but it's important to understand that you really have multiple channels, and hence multiple two node clusters. With JBoss AS, it's the services that form clusters, not the servers.

On the same network, you may have different sets of servers whose services wish to cluster. *Figure 2.1, “Clusters and server nodes”* shows an example network of JBoss server instances divided into three sets, with the third set only having one node. This sort of topology can be set up simply by configuring the AS instances such that within a set of nodes meant to form a cluster the channel configurations and names match while they differ from any other channels on the same network. The AS tries to make this is easy as possible, such that servers that are meant to cluster only need to have the same values passed on the command line to the `-g` (partition name) and `-u` (multicast address) startup switches. For each set of servers, different values should be chosen. The sections on “JGroups Configuration” and “Isolating JGroups Channels” cover in detail how to configure the AS such that desired peers find each other and unwanted peers do not.

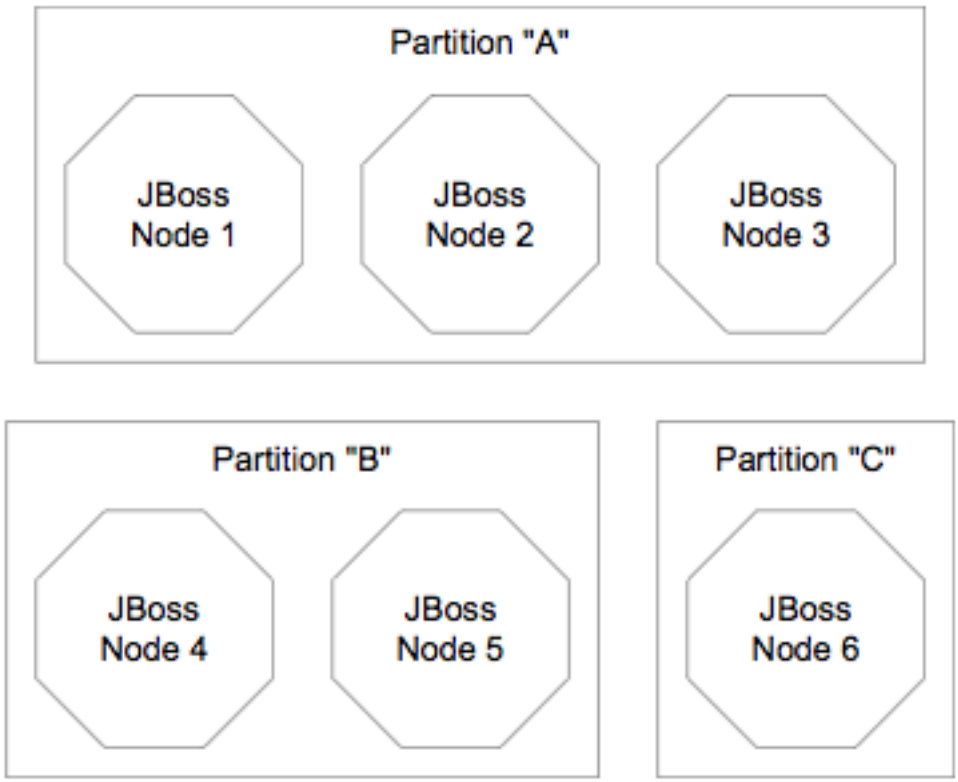

Figure 2.1. Clusters and server nodes

2.2. Service Architectures

The clustering topography defined by the JGroups configuration on each node is of great importance to system administrators. But for most application developers, the greater concern is probably the cluster architecture from a client application's point of view. Two basic clustering architectures are used with JBoss AS: client-side interceptors (a.k.a smart proxies or stubs) and external load balancers. Which architecture your application will use will depend on what type of client you have.

2.2.1. Client-side interceptor architecture

Most remote services provided by the JBoss application server, including JNDI, EJB, JMS, RMI and JBoss Remoting, require the client to obtain (e.g., to look up and download) a remote proxy object. The proxy object is generated by the server and it implements the business interface of the service. The client then makes local method calls against the proxy object. The proxy automatically routes the call across the network where it is invoked against service objects managed in the server. The proxy object figures out how to find the appropriate server node, marshal call parameters, un-marshall call results, and return the result to the caller client. In a clustered environment, the server-generated proxy object includes an interceptor that understands how to route calls to multiple nodes in the cluster.

The proxy's clustering logic maintains up-to-date knowledge about the cluster. For instance, it knows the IP addresses of all available server nodes, the algorithm to distribute load across nodes (see next section), and how to failover the request if the target node not available. As part of handling each service request, if the cluster topology has changed the server node updates the proxy with the latest changes in the cluster. For instance, if a node drops out of the cluster, each proxy is updated with the new topology the next time it connects to any active node in the cluster. All the manipulations done by the proxy's clustering logic are transparent to the client application. The client-side interceptor clustering architecture is illustrated in *Figure 2.2, "The client-side interceptor (proxy) architecture for clustering"*.

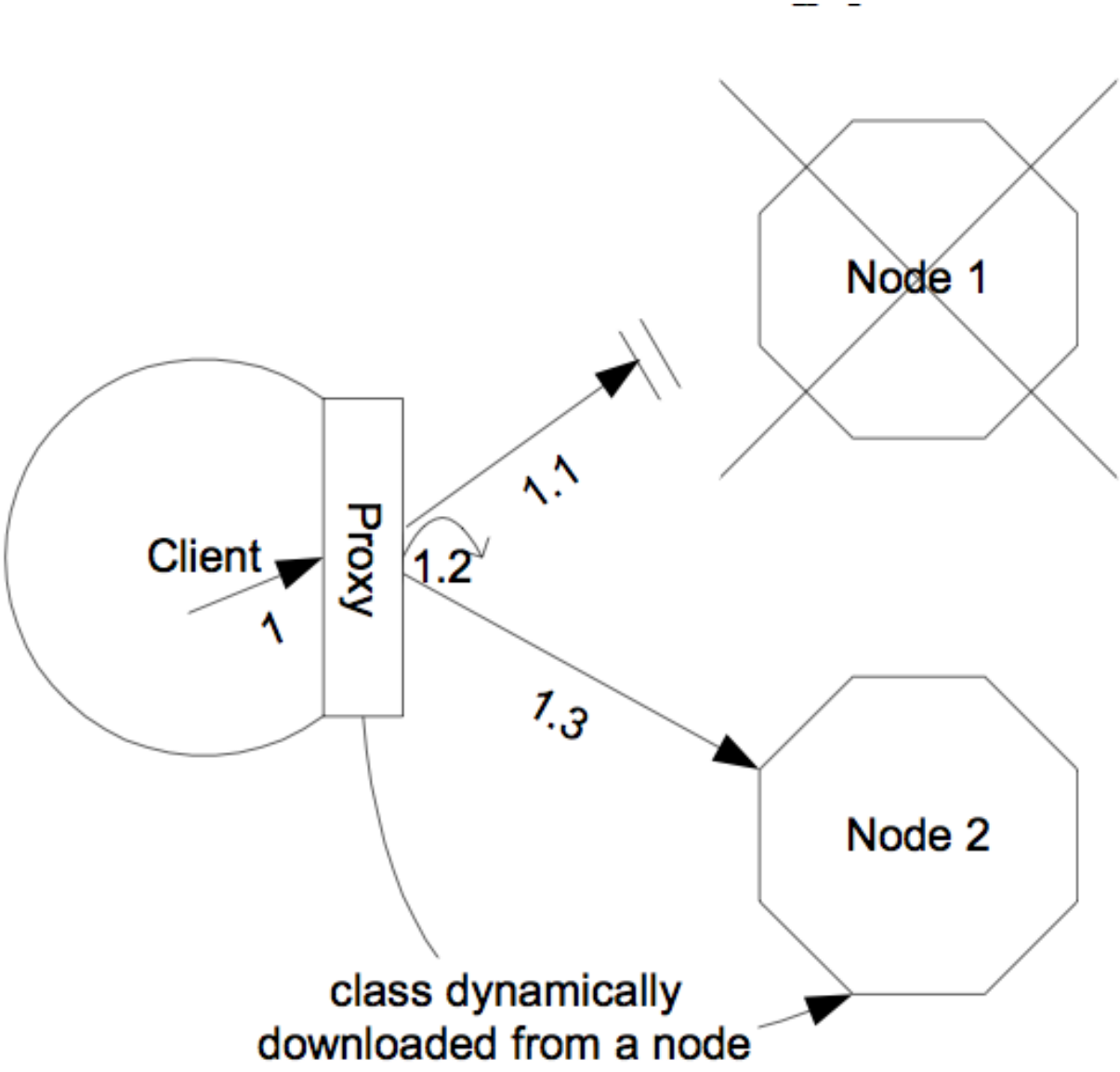

Figure 2.2. The client-side interceptor (proxy) architecture for clustering

2.2.2. External Load Balancer Architecture

The HTTP-based JBoss services do not require the client to download anything. The client (e.g., a web browser) sends in requests and receives responses directly over the wire using the HTTP protocol). In this case, an external load balancer is required to process all requests and dispatch them to server nodes in the cluster. The client only needs to know how to contact the load balancer; it has no knowledge of the JBoss AS instances behind the load balancer. The load balancer is logically part of the cluster, but we refer to it as “external” because it is not running in the same process as either the client or any of the JBoss AS instances. It can be implemented either in software or hardware. There are many vendors of hardware load balancers; the mod_jk Apache module is an excellent example of a software load balancer. An external load balancer implements its own mechanism for understanding the cluster configuration and provides its own load balancing and failover policies. The external load balancer clustering architecture is illustrated in *Figure 2.3, “The external load balancer architecture for clustering”*.

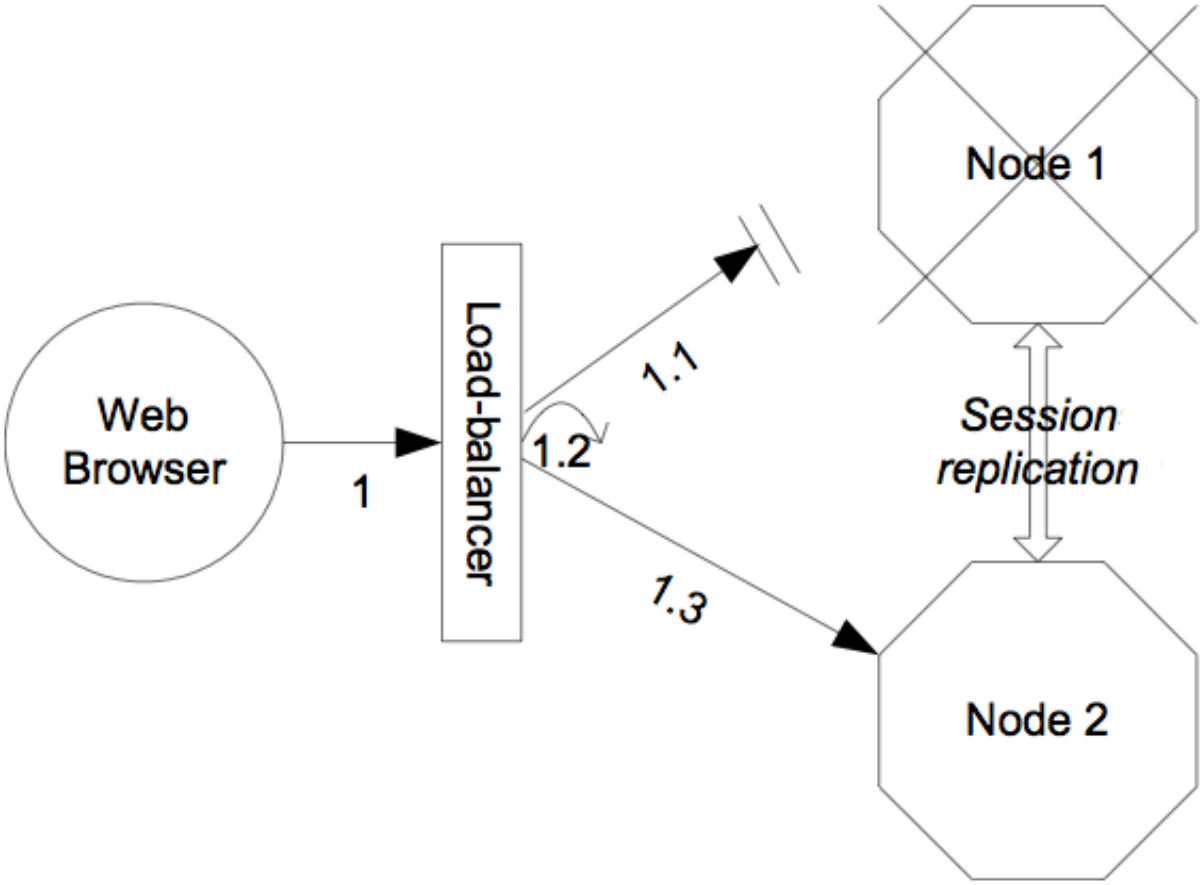

Figure 2.3. The external load balancer architecture for clustering

A potential problem with an external load balancer architecture is that the load balancer itself may be a single point of failure. It needs to be monitored closely to ensure high availability of the entire cluster's services.

2.3. Load-Balancing Policies

Both the JBoss client-side interceptor (stub) and load balancer use load balancing policies to determine to which server node a new request should be sent. In this section, let's go over the load balancing policies available in JBoss AS.

2.3.1. Client-side interceptor architecture

In JBoss AS 5, the following load balancing options are available when the client-side interceptor architecture is used. The client-side stub maintains a list of all nodes providing the target service; the job of the load balance policy is to pick a node from this list for each request. Each policy has two implementation classes, one meant for use by legacy services like EJB2 that use the legacy detached invoker architecture, and the other meant for services like EJB3 that use AOP-based invocations.

- Round-Robin: each call is dispatched to a new node, proceeding sequentially through the list of nodes. The first target node is randomly selected from the list. Implemented by `org.jboss.ha.framework.interfaces.RoundRobin` (legacy) and `org.jboss.ha.client.loadbalance.RoundRobin` (EJB3).
- Random-Robin: for each call the target node is randomly selected from the list. Implemented by `org.jboss.ha.framework.interfaces.RoundRobin` (legacy) and `org.jboss.ha.client.loadbalance.RoundRobin` (EJB3).
- First Available: one of the available target nodes is elected as the main target and is thereafter used for every call; this elected member is randomly chosen from the list of members in the cluster. When the list of target nodes changes (because a node starts or dies), the policy will choose a new target node unless the currently elected node is still available. Each client-side proxy elects its own target node independently of the other proxies, so if a particular client downloads two proxies for the same target service (e.g., an EJB), each proxy will independently pick its target. This is an example of a policy that provides “session affinity” or “sticky sessions”, since the target node does not change once established. Implemented by `org.jboss.ha.framework.interfaces.FirstAvailable` (legacy) and `org.jboss.ha.client.loadbalance.aop.FirstAvailable` (EJB3).
- First Available Identical All Proxies: has the same behaviour as the "First Available" policy but the elected target node is shared by all proxies in the same client-side VM that are associated with the same target service. So if a particular client downloads two proxies for the same target service (e.g. an EJB), each proxy will use the same target. Implemented by `org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies` (legacy) and `org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies` (EJB3).

New in JBoss 5 are a set of "TransactionSticky" load balance policies. These extend the standard policies above to add behavior such that all invocations that occur within the scope of a transaction

are routed to the same node (if that node still exists). These are based on the legacy detached invoker architecture, so they are not available for AOP-based services like EJB3.

- Transaction-Sticky Round-Robin: Transaction-sticky variant of Round-Robin. Implemented by `org.jboss.ha.framework.interfaces.TransactionStickyRoundRobin`.
- Transaction-Sticky Random-Robin: Transaction-sticky variant of Random-Robin. Implemented by `org.jboss.ha.framework.interfaces.TransactionStickyRandomRobin`.
- Transaction-Sticky First Available: Transaction-sticky variant of First Available. Implemented by `org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailable`.
- Transaction-Sticky First Available Identical All Proxies: Transaction-sticky variant of First Available Identical All Proxies. Implemented by `org.jboss.ha.framework.interfaces.TransactionStickyFirstAvailableIdenticalAllProxies`.

Each of the above is an implementation of a simple interface; users are free to write their own implementations if they need some special behavior. In later sections we'll see how to configure the load balance policies used by different services.

2.3.2. External load balancer architecture

As noted above, an external load balancer provides its own load balancing capabilities. What capabilities are supported depends on the provider of the load balancer. The only JBoss requirement is that the load balancer support “session affinity” (a.k.a. “sticky sessions”). With session affinitiy enabled, once the load balancer routes a request from a client to node A and the server initiates a session, all future requests associated with that session must be routed to node A, so long as node A is available.

Clustering Building Blocks

The clustering features in JBoss AS are built on top of lower level libraries that provide much of the core functionality. *Figure 3.1, "The JBoss AS clustering architecture"* shows the main pieces:

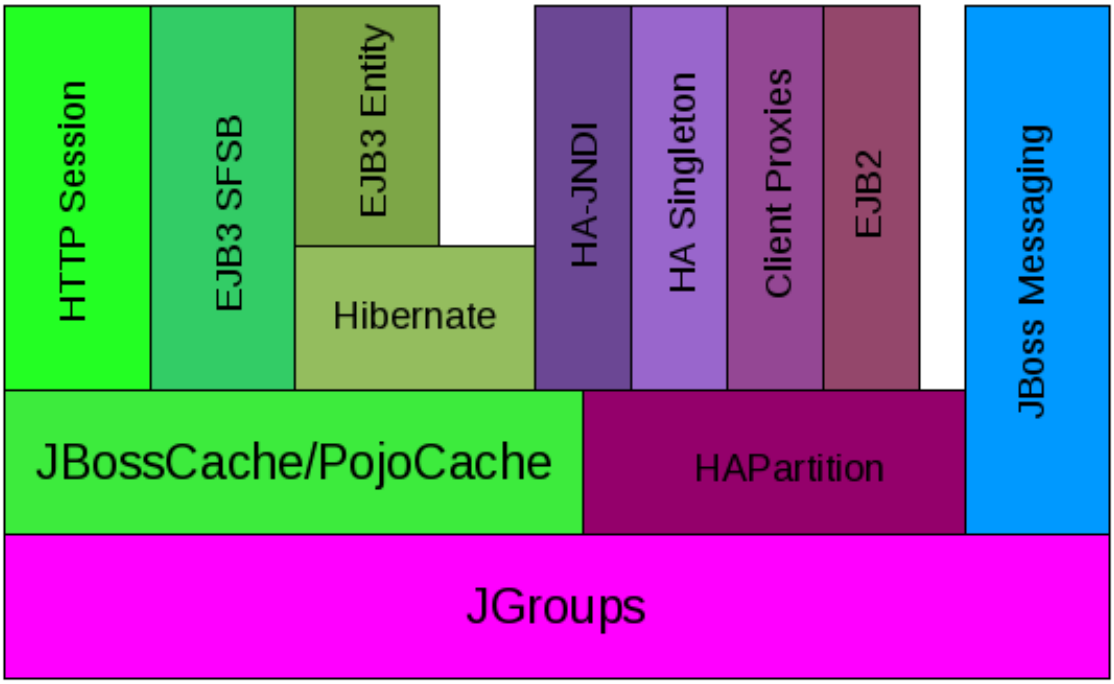

Figure 3.1. The JBoss AS clustering architecture

JGroups is a toolkit for reliable point-to-point and point-to-multipoint communication. JGroups is used for all clustering-related communications between nodes in a JBoss AS cluster. See *Section 3.1, "Group Communication with JGroups"* for more on how JBoss AS uses JGroups.

JBoss Cache is a highly flexible clustered transactional caching library. Many AS clustering services need to cache some state in memory while 1) ensuring for high availability purposes that a backup copy of that state is available on another node if it can't otherwise be recreated (e.g. the contents of a web session) and 2) ensuring that the data cached on each node in the cluster is consistent. JBoss Cache handles these concerns for most JBoss AS clustered services. JBoss Cache uses JGroups to handle its group communication requirements. **POJO Cache** is an extension of the core JBoss Cache that JBoss AS uses to support fine-grained replication of clustered web session state. See *Section 3.2, "Distributed Caching with JBoss Cache"* for more on how JBoss AS uses JBoss Cache and POJO Cache.

HAPartition is an adapter on top of a JGroups channel that allows multiple services to use the channel. HAPartition also supports a distributed registry of which HAPartition-based services are running on which cluster members. It provides notifications to interested listeners when the cluster

membership changes or the clustered service registry changes. See *Section 3.3, "The HAPartition Service"* for more details on HAPartition.

The other higher level clustering services make use of JBoss Cache or HAPartition, or, in the case of HA-JNDI, both. The exception to this is JBoss Messaging's clustering features, which interact with JGroups directly.

3.1. Group Communication with JGroups

JGroups provides the underlying group communication support for JBoss AS clusters. Services deployed on JBoss AS which need group communication with their peers will obtain a JGroups `Channel` and use it to communicate. The `Channel` handles such tasks as managing which nodes are members of the group, detecting node failures, ensuring lossless, first-in-first-out delivery of messages to all group members, and providing flow control to ensure fast message senders cannot overwhelm slow message receivers.

The characteristics of a JGroups `Channel` are determined by the set of *protocols* that compose it. Each protocol handles a single aspect of the overall group communication task; for example the `UDP` protocol handles the details of sending and receiving UDP datagrams. A `Channel` that uses the `UDP` protocol is capable of communicating with UDP unicast and multicast; alternatively one that uses the `TCP` protocol uses TCP unicast for all messages. JGroups supports a wide variety of different protocols (see *Section 10.1, "Configuring a JGroups Channel's Protocol Stack"* for details), but the AS ships with a default set of channel configurations that should meet most needs.

By default, UDP multicast is used by all JGroups channels used by the AS (except for one TCP-based channel used by JBoss Messaging).

3.1.1. The Channel Factory Service

A significant difference in JBoss AS 5 versus previous releases is that JGroups Channels needed by clustering services (e.g. a channel used by a distributed HttpSession cache) are no longer configured in detail as part of the consuming service's configuration, and are no longer directly instantiated by the consuming service. Instead, a new `ChannelFactory` service is used as a registry for named channel configurations and as a factory for `Channel` instances. A service that needs a channel requests the channel from the `ChannelFactory`, passing in the name of the desired configuration.

The ChannelFactory service is deployed in the `server/all/deploy/cluster/jgroups-channelfactory.sar`. On startup the ChannelFactory service parses the `server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file, which includes various standard JGroups configurations identified by name (e.g "udp" or "tcp"). Services needing a channel access the channel factory and ask for a channel with a particular named configuration.

3.1.1.1. Standard Protocol Stack Configurations

The standard protocol stack configurations that ship with AS 5 are described below. Note that not all of these are actually used; many are included as a convenience to users who may wish to alter

the default server configuration. The configurations actually used in a stock AS 5 **all** config are `udp`, `jbm-control` and `jbm-data`, with all clustering services other than JBoss Messaging using `udp`.

- **udp**

  UDP multicast based stack meant to be shared between different channels. Message bundling is disabled, as it can add latency to synchronous group RPCs. Services that only make asynchronous RPCs (e.g. JBoss Cache configured for REPL_ASYNC) and do so in high volume may be able to improve performance by configuring their cache to use the `udp-async` stack below. Services that only make synchronous RPCs (e.g. JBoss Cache configured for REPL_SYNC or INVALIDATION_SYNC) may be able to improve performance by using the `udp-sync` stack below, which does not include flow control.

- **udp-async**

  Same as the default `udp` stack above, except message bundling is enabled in the transport protocol (`enable_bundling=true`). Useful for services that make high-volume asynchronous RPCs (e.g. high volume JBoss Cache instances configured for REPL_ASYNC) where message bundling may improve performance.

- **udp-sync**

  UDP multicast based stack, without flow control and without message bundling. This can be used instead of `udp` if (1) synchronous calls are used and (2) the message volume (rate and size) is not that large. Don't use this configuration if you send messages at a high sustained rate, or you might run out of memory.

- **tcp**

  TCP based stack, with flow control and message bundling. TCP stacks are usually used when IP multicasting cannot be used in a network (e.g. routers discard multicast).

- **tcp-sync**

  TCP based stack, without flow control and without message bundling. TCP stacks are usually used when IP multicasting cannot be used in a network (e.g.routers discard multicast). This configuration should be used instead of `tcp` above when (1) synchronous calls are used and (2) the message volume (rate and size) is not that large. Don't use this configuration if you send messages at a high sustained rate, or you might run out of memory.

- **jbm-control**

  Stack optimized for the JBoss Messaging Control Channel. By default uses the same UDP transport protocol config as is used for the default 'udp' stack defined above. This allows the JBoss Messaging Control Channel to use the same sockets, network buffers and thread pools as are used by the other standard JBoss AS clustered services (see *Section 3.1.2, "The JGroups Shared Transport"*.

- **jbm-data**

  Stack optimized for the JBoss Messaging Data Channel. TCP-based

You can add a new stack configuration by adding a new `stack` element to the `server/all/deploy/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file. You can alter the behavior of an existing configuration by editing this file. Before doing this though, have a look at the other standard configurations the AS ships; perhaps one of those meets your needs. Also, please note that before editing a configuration you should understand what services are using that configuration; make sure the change you are making is appropriate for all affected services. If the change isn't appropriate for a particular service, perhaps its better to create a new configuration and change some services to use that new configuration.

It's important to note that if several services request a channel with the same configuration name from the ChannelFactory, they are not handed a reference to the same underlying Channel. Each receives its own Channel, but the channels will have an identical configuration. A logical question is how those channels avoid forming a group with each other if each, for example, is using the same multicast address and port. The answer is that when a consuming service connects its Channel, it passes a unique-to-that-service `cluster_name` argument to the `Channel.connect(String cluster_name)` method. The Channel uses that `cluster_name` as one of the factors that determine whether a particular message received over the network is intended for it.

3.1.2. The JGroups Shared Transport

As the number of JGroups-based clustering services running in the AS has risen over the years, the need to share the resources (particularly sockets and threads) used by these channels became a glaring problem. A stock AS 5 **all** config will connect 4 JGroups channels during startup, and a total of 7 or 8 will be connected if distributable web apps, clustered EJB3 SFSBs and a clustered JPA/Hibernate second level cache are all used. So many channels can consume a lot of resources, and can be a real configuration nightmare if the network environment requires configuration to ensure cluster isolation.

Beginning with AS 5, JGroups supports sharing of transport protocol instances between channels. A JGroups channel is composed of a stack of individual protocols, each of which is responsible for one aspect of the channel's behavior. A transport protocol is a protocol that is responsible for actually sending messages on the network and receiving them from the network. The resources that are most desirable for sharing (sockets and thread pools) are managed by the transport protocol, so sharing a transport protocol between channels efficiently accomplishes JGroups resource sharing.

To configure a transport protocol for sharing, simply add a `singleton_name="someName"` attribute to the protocol's configuration. All channels whose transport protocol config uses the same `singleton_name` value will share their transport. All other protocols in the stack will not be shared. The following illustrates 4 services running in a VM, each with its own channel. Three of the services are sharing a transport; the fourth is using its own transport.

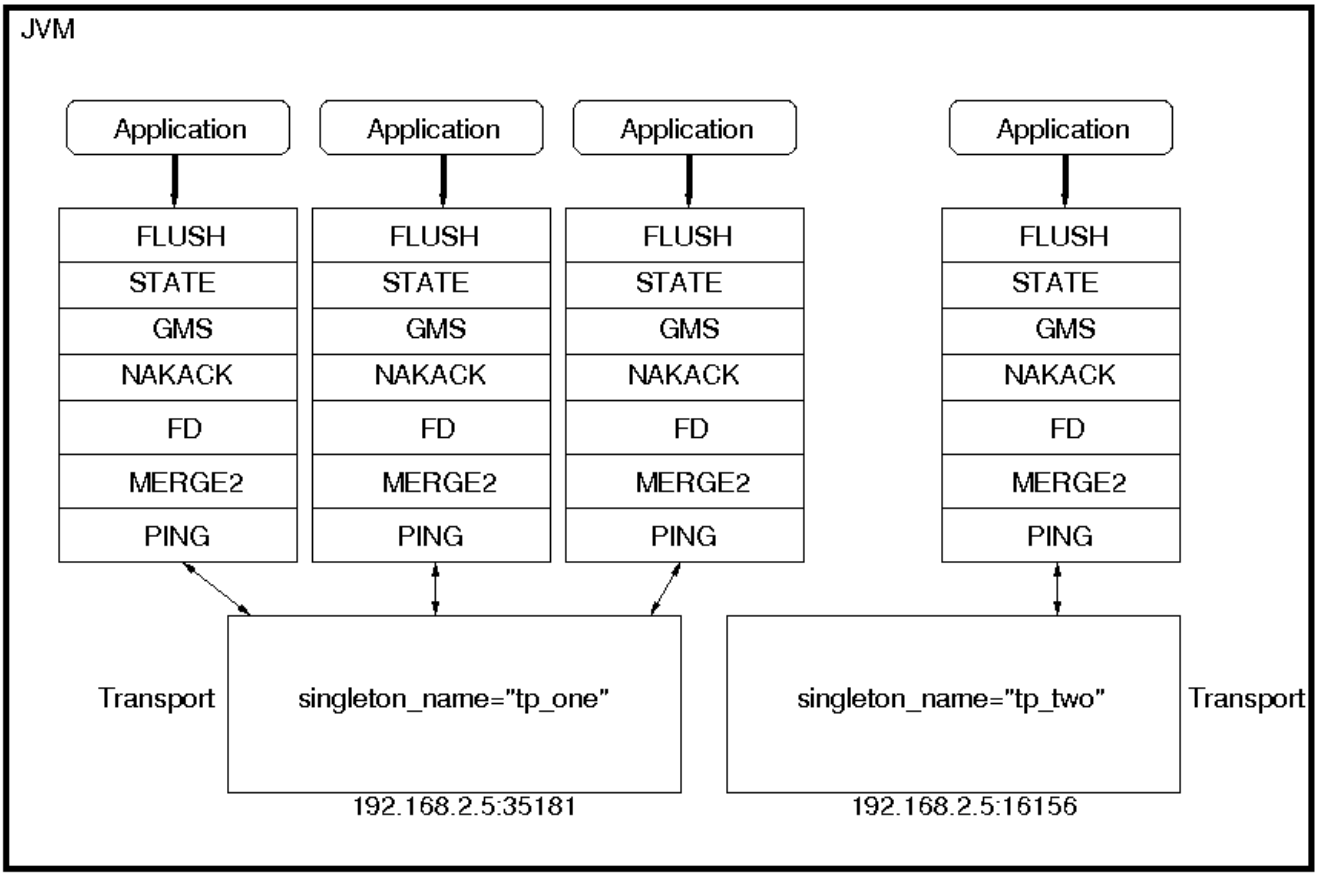

Figure 3.2. Services using a Shared Transport

The protocol stack configurations used by the AS 5 ChannelFactory all have a singleton_name configured. In fact, if you add a stack to the ChannelFactory that doesn't include a `singleton_name`, before creating any channels for that stack, the ChannelFactory will synthetically create a `singleton_name` by concatenating the stack name to the string "unnamed_", e.g. unnamed_customStack.

3.2. Distributed Caching with JBoss Cache

JBoss Cache is a fully featured distributed cache framework that can be used in any application server environment or standalone. JBoss Cache provides the underlying distributed caching support used by many of the standard clustered services in a JBoss AS cluster, including:

- Replication of clustered webapp sessions.
- Replication of clustered EJB3 Stateful Session beans.
- Clustered caching of JPA and Hibernate entities.
- Clustered Single Sign-On.
- The HA-JNDI replicated tree.
- DistributedStateService

Users can also create their own JBoss Cache and POJO Cache instances for custom use by their applications, see *Chapter 11, JBoss Cache Configuration and Deployment* for more on this.

3.2.1. The JBoss AS CacheManager Service

Many of the standard clustered services in JBoss AS use JBoss Cache to maintain consistent state across the cluster. Different services (e.g. web session clustering or second level caching of JPA/Hibernate entities) use different JBoss Cache instances, with each cache configured to meet the needs of the service that uses it. In AS 4, each of these caches was independently deployed in the `deploy/` directory, which had a number of disadvantages:

- Caches that end user applications didn't need were deployed anyway, with each creating an expensive JGroups channel. For example, even if there were no clustered EJB3 SFSBs, a cache to store them was started.
- Caches are internal details of the services that use them. They shouldn't be first-class deployments.
- Services would find their cache via JMX lookups. Using JMX for purposes other exposing management interfaces is just not the JBoss 5 way.

In JBoss 5, the scattered cache deployments have been replaced with a new **CacheManager** service, deployed via the `JBOSS_HOME/server/all/deploy/cluster/jboss-cache-manager.sar`. The CacheManager is a factory and registry for JBoss Cache instances. It is configured with a set of named JBoss Cache configurations. Services that need a cache ask the cache manager for the cache by name; the cache manager creates the cache (if not already created) and returns it. The cache manager 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 cache manager. The cache manager keeps track of how many services are using each cache, and will stop and destroy the cache when all services have released it.

3.2.1.1. Standard Cache Configurations

The following standard JBoss Cache configurations ship with JBoss AS 5. You can add others to suit your needs, or edit these configurations to adjust cache behavior. Additions or changes are done by editing the `deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml` file (see *Section 11.2.1, "Deployment Via the CacheManager Service"* for details). Note however that these configurations are specifically optimized for their intended use, and except as specifically noted in the documentation chapters for each service in this guide, it is not advisable to change them.

- **standard-session-cache**

  Standard cache used for web sessions.

- **field-granularity-session-cache**

  Standard cache used for FIELD granularity web sessions.

- **sfsb-cache**

  Standard cache used for EJB3 SFSB caching.

- **ha-partition**

  Used by web tier Clustered Single Sign-On, HA-JNDI, Distributed State.

- **mvcc-entity**

  A config appropriate for JPA/Hibernate entity/collection caching that uses JBC's MVCC locking (see notes below).

- **optimistic-entity**

  A config appropriate for JPA/Hibernate entity/collection caching that uses JBC's optimistic locking (see notes below).

- **pessimistic-entity**

  A config appropriate for JPA/Hibernate entity/collection caching that uses JBC's pessimistic locking (see notes below).

- **mvcc-entity-repeatable**

  Same as "mvcc-entity" but uses JBC's REPEATABLE_READ isolation level instead of READ_COMMITTED (see notes below).

- **pessimistic-entity-repeatable**

  Same as "pessimistic-entity" but uses JBC's REPEATABLE_READ isolation level instead of READ_COMMITTED (see notes below).

- **local-query**

  A config appropriate for JPA/Hibernate query result caching. Does not replicate query results. DO NOT store the timestamp data Hibernate uses to verify validity of query results in this cache.

- **replicated-query**

  A config appropriate for JPA/Hibernate query result caching. Replicates query results. DO NOT store the timestamp data Hibernate uses to verify validity of query result in this cache.

- **timestamps-cache**

  A config appropriate for the timestamp data cached as part of JPA/Hibernate query result caching. A replicated timestamp cache is required if query result caching is used, even if the query results themselves use a non-replicating cache like `local-query`.

- **mvcc-shared**

  A config appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode

REPL_SYNC, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Maintained for backwards compatibility reasons, as a shared cache was the only option in JBoss 4. Uses JBC's MVCC locking.

- **optimistic-shared**

  A config appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode REPL_SYNC, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Maintained for backwards compatibility reasons, as a shared cache was the only option in JBoss 4. Uses JBC's optimistic locking.

- **pessimistic-shared**

  A config appropriate for a cache that's shared for JPA/Hibernate entity, collection, query result and timestamp caching. Not an advised configuration, since it requires cache mode REPL_SYNC, which is the least efficient mode. Also requires a full state transfer at startup, which can be expensive. Maintained for backwards compatibility reasons, as a shared cache was the only option in JBoss 4. Uses JBC's pessimistic locking.

- **mvcc-shared-repeatable**

  Same as "mvcc-shared" but uses JBC's REPEATABLE_READ isolation level instead of READ_COMMITTED (see notes below).

- **pessimistic-shared-repeatable**

  Same as "pessimistic-shared" but uses JBC's REPEATABLE_READ isolation level instead of READ_COMMITTED. (see notes below).

> **Note**
>
> For more on JBoss Cache's locking schemes, see *Section 11.1.4, "Concurrent Access"*)

> **Note**
>
> For JPA/Hibernate second level caching, REPEATABLE_READ is only useful if the application evicts/clears entities from the EntityManager/Hibernate Session and then expects to repeatably re-read them in the same transaction. Otherwise, the Session's internal cache provides a repeatable-read semantic.

3.2.1.2. Cache Configuration Aliases

The CacheManager also supports aliasing of caches; i.e. allowing caches registered under one name to be looked up under a different name. Aliasing is useful for sharing caches between

services whose configuration may specify different cache config names. It's also useful for supporting legacy EJB3 application configurations ported over from AS 4.

Aliases can be configured by editing the "CacheManager" bean in the `jboss-cache-manager-jboss-beans.xml` file. The following redacted config shows the standard aliases in AS 5.0.0.GA:

```
<bean name="CacheManager" class="org.jboss.ha.cachemanager.CacheManager">

   . . .

   <!-- Aliases for cache names. Allows caches to be shared across
        services that may expect different cache config names. -->
   <property name="configAliases">
     <map keyClass="java.lang.String" valueClass="java.lang.String">
       <!-- Use the HAPartition cache for ClusteredSSO caching -->
       <entry>
         <key>clustered-sso</key>
         <value>ha-partition</value>
       </entry>
       <!-- Handle the legacy name for the EJB3 SFSB cache -->
       <entry>
         <key>jboss.cache:service=EJB3SFSBClusteredCache</key>
         <value>sfsb-cache</value>
       </entry>
       <!-- Handle the legacy name for the EJB3 Entity cache -->
       <entry>
         <key>jboss.cache:service=EJB3EntityTreeCache</key>
         <value>mvcc-shared</value>
       </entry>
     </map>
   </property>

   . . .

</bean>
```

3.3. The HAPartition Service

HAPartition is a general purpose service used for a variety of tasks in AS clustering. At its core, it is an abstraction built on top of a JGroups `Channel` that provides support for making/receiving RPC invocations on/from one or more cluster members. HAPartition allows services that use it 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`. HAPartition also supports a distributed registry of which clustering services are running on which cluster members. It provides notifications to interested listeners when the cluster membership changes or the clustered service registry changes. HAPartition forms the core of many of the clustering services we'll be discussing in the rest of this guide, including smart client-side clustered proxies, EJB 2 SFSB replication and entity cache management, farming, HA-JNDI and HA singletons. Custom services can also make use of HAPartition.

The following snippet shows the `HAPartition` service definition packaged with the standard JBoss AS distribution. This configuration can be found in the `server/all/deploy/cluster/hapartition-jboss-beans.xml` file.

```
<bean name="HAPartitionCacheHandler"
      class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
      <property name="cacheManager"><inject bean="CacheManager"/></property>
      <property name="cacheConfigName">ha-partition</property>
</bean>

<bean name="HAPartition" class="org.jboss.ha.framework.server.ClusterPartition">

   <depends>jboss:service=Naming</depends>
   <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>

   <!-- ClusterPartition requires a Cache for state management -->
  <property name="cacheHandler"><inject bean="HAPartitionCacheHandler"/></property>
   <!-- Name of the partition being built -->
   <property name="partitionName">${jboss.partition.name:DefaultPartition}</property>
   <!-- The address used to determine the node name -->
   <property name="nodeAddress">${jboss.bind.address}</property>
   <!-- Max time (in ms) to wait for state transfer to complete. -->
   <property name="stateTransferTimeout">30000</property>
   <!-- Max time (in ms) to wait for RPC calls to complete. -->
   <property name="methodCallTimeout">60000</property>

   <!-- Optionally provide a thread source to allow async connect of our channel -->
   <property name="threadPool">
      <inject bean="jboss.system:service=ThreadPool"/>
   </property>

   <property name="distributedStateImpl">
      <bean name="DistributedState"
            class="org.jboss.ha.framework.server.DistributedStateImpl">
         <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>
         <property name="cacheHandler">
```

```
          <inject bean="HAPartitionCacheHandler"/>
        </property>
      </bean>
    </property>
</bean>
```

Much of the above is boilerplate; below we'll touch on the key points relevant to end users. There are two beans defined above, the `HAPartitionCacheHandler` and the `HAPartition` itself.

The `HAPartition` bean itself exposes the following configuration properties:

- **partitionName** specifies the name of the cluster. Its default value is `DefaultPartition`. Use the `-g` (a.k.a. --partition) command line switch to set this value at JBoss startup.
- **nodeAddress** is unused and can be ignored.
- **stateTransferTimeout** specifies the timeout (in milliseconds) for initial application state transfer. State transfer refers to the process of obtaining a serialized copy of initial application state from other already-running cluster members at service startup. Its default value is `30000`.
- **methodCallTimeout** specifies the timeout (in milliseconds) for obtaining responses to group RPCs from the other cluster members. Its default value is `60000`.

The `HAPartitionCacheHandler` is a small utility service that helps the HAPartition integrate with JBoss Cache (see *Section 3.2.1, "The JBoss AS CacheManager Service"*). HAPartition exposes a child service called DistributedState (see *Section 3.3.2, "DistributedState Service"*) that uses JBoss Cache; the `HAPartitionCacheHandler` helps ensure consistent configuration between the JGroups `Channel` used by Distributed State's cache and the one used directly by HAPartition.

- **cacheConfigName** the name of the JBoss Cache configuration to use for the HAPartition-related cache. Indirectly, this also specifies the name of the JGroups protocol stack configuration HAPartition should use. See *Section 11.1.5, "JGroups Integration"* for more on how the JGroups protocol stack is configured.

In order for nodes to form a cluster, they must have the exact same `partitionName` and the `HAPartitionCacheHandler`'s `cacheConfigName` must specify an identical JBoss Cache configuration. Changes in either element on some but not all nodes would prevent proper clustering behavior.

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (i.e., `http://hostname:8080/jmx-console/`) and then clicking on the `jboss:service=HAPartition,partition=DefaultPartition` MBean (change the MBean name to reflect your partitionr name if you use the -g startup switch). A list of IP addresses for the current cluster members is shown in the CurrentView field.

> **Note**
>
> While it is technically possible to put a JBoss server instance into multiple HAPartitions at the same time, this practice is generally not recommended, as it increases management complexity.

3.3.1. DistributedReplicantManager Service

The `DistributedReplicantManager` (DRM) service is a component of the HAPartition service made available to HAPartition users via the `HAPartition.getDistributedReplicantManager()` method. Generally speaking, JBoss AS users will not directly make use of the DRM; we discuss it here as an aid to those who want a deeper understanding of how AS clustering internals work.

The DRM is a distributed registry that allows HAPartition users to register objects under a given key, making available to callersthe set of objects registered under that key by the various members of t he cluster. The DRM also provides a notification mechanism so interested listeners can be notified when the contents of the registry changes.

There are two main usages for the DRM in JBoss AS:

- **Clustered Smart Proxies**

  Here the keys are the names of the various services that need a clustered smart proxy (see *Section 2.2.1, "Client-side interceptor architecture"*, e.g. the name of a clustered EJB. The value object each node stores in the DRM is known as a "target". It's something a smart proxy's transport layer can use to contact the node (e.g. an RMI stub, an HTTP URL or a JBoss Remoting `InvokerLocator`). The factory that builds clustered smart proxies accesses the DRM to get the set of "targets" that should be injected into the proxy to allow it to communicate with all the nodes in a cluster.

- **HASingleton**

  Here the keys are the names of the various services that need to function as High Availablity Singletons (see *???*). The value object each node stores in the DRM is simply a String that acts as a token to indicate that the node has the service deployed, and thus is a candidate to become the "master" node for the HA singleton service.

In both cases, the key under which objects are registered identifies a particular clustered service. It is useful to understand that every node in a cluster doesn't have to register an object under every key. Only services that are deployed on a particular node will register something under that service's key, and services don't have to be deployed homogeneously across the cluster. The DRM is thus useful as a mechanism for understanding a service's "topology" around the cluster -- which nodes have the service deployed.

3.3.2. DistributedState Service

The `DistributedState` service is a legacy component of the HAPartition service made available to HAPartition users via the `HAPartition.getDistributedState()` method. This service provides coordinated management of arbitary application state around the cluster. It is supported for backwards compatibility reasons, but new applications should not use it; they should use the much more sophisticated JBoss Cache instead.

In JBoss 5 the `DistributedState` service actually delegates to an underlying JBoss Cache instance.

3.3.3. Custom Use of HAPartition

Custom services can also use make use of HAPartition to handle interactions with the cluster. Generally the easiest way to do this is to extend the `org.jboss.ha.framework.server.HAServiceImpl` base class, or the `org.jboss.ha.jxm.HAServiceMBeanSupport` class if JMX registration and notification support are desired.

Clustered Deployment Options

4.1. Clustered Singleton Services

A clustered singleton service (also known as an HA singleton) is a service that is deployed on multiple nodes in a cluster, but is providing its service on only one of the nodes. The node running the singleton service is typically called the master node. When the master fails or is shut down, another master is selected from the remaining nodes and the service is restarted on the new master. Thus, other than a brief interval when one master has stopped and another has yet to take over, the service is always being provided by one but only one node.

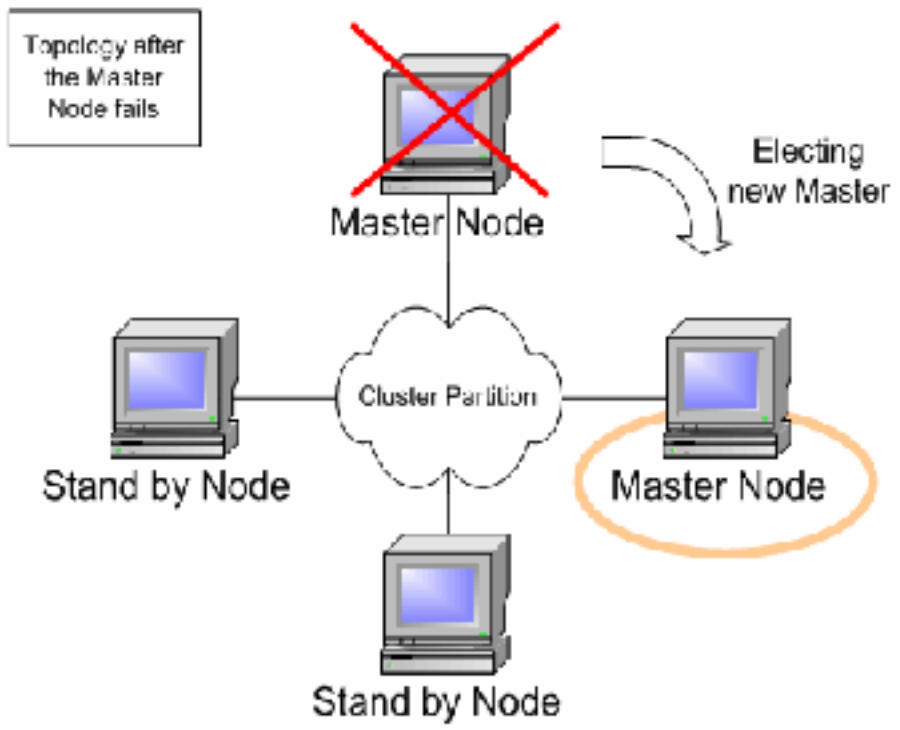

Figure 4.1. Topology after the Master Node fails

4.1.1. HASingleton Deployment Options

The JBoss Application Server (AS) provides support for a number of strategies for helping you deploy clustered singleton services. In this section we will explore the different strategies. All of the strategies are built on top of the HAPartition service described in the introduction. They rely on the `HAPartition` to provide notifications when different nodes in the cluster start and stop; based on those notifications each node in the cluster can independently (but consistently) determine if it is now the master node and needs to begin providing a service.

4.1.1.1. HASingletonDeployer service

The simplest and most commonly used strategy for deploying an HA singleton is to take an ordinary deployment (war, ear, jar, whatever you would normally put in deploy) and deploy it in the `$JBOSS_HOME/server/all/deploy-hasingleton` directory instead of in `deploy`. The `deploy-hasingleton` directory does not lie under `deploy` nor `farm` directories, so its contents are not automatically deployed when an AS instance starts. Instead, deploying the contents of this directory is the responsibility of a special service, the `HASingletonDeployer` bean (which itself is deployed via the deploy/deploy-hasingleton-jboss-beans.xml file.) The HASingletonDeployer service is itself an HA Singleton, one whose provided service, when it becomes master, is to deploy the contents of deploy-hasingleton; and whose service, when it stops being the master (typically at server shutdown), is to undeploy the contents of `deploy-hasingleton`.

So, by placing your deployments in `deploy-hasingleton` you know that they will be deployed only on the master node in the cluster. If the master node cleanly shuts down, they will be cleanly undeployed as part of shutdown. If the master node fails or is shut down, they will be deployed on whatever node takes over as master.

Using deploy-hasingleton is very simple, but it does have two drawbacks:

- There is no hot-deployment feature for services in `deploy-hasingleton` . Redeploying a service that has been deployed to `deploy-hasingleton` requires a server restart.
- If the master node fails and another node takes over as master, your singleton service needs to go through the entire deployment process before it will be providing services. Depending on the complexity of your service's deployment, and the extent of startup activity in which it engages, this could take a while, during which time the service is not being provided.

4.1.1.2. POJO deployments using HASingletonController

If your service is a POJO (i.e., not a J2EE deployment like an ear or war or jar), you can deploy it along with a service called an HASingletonController in order to turn it into an HA singleton. It is the job of the HASingletonController to work with the HAPartition service to monitor the cluster and determine if it is now the master node for its service. If it determines it has become the master node, it invokes a method on your service telling it to begin providing service. If it determines it is no longer the master node, it invokes a method on your service telling it to stop providing service. Let's walk through an illustration.

First, we have a POJO that we want to make an HA singleton. The only thing special about it is it needs to expose a public method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```
public interface HASingletonExampleMBean
{
   boolean isMasterNode();
}
```

```
public class HASingletonExample implements HASingletonExampleMBean
{
  private boolean isMasterNode = false;

  public boolean isMasterNode()
  {
    return isMasterNode;
  }

  public void startSingleton()
  {
    isMasterNode = true;
  }

  public void stopSingleton()
  {
    isMasterNode = false;
  }
}
```

We used `startSingleton` and `stopSingleton` in the above example, but you could name the methods anything.

Next, we deploy our service, along with an HASingletonController to control it, most likely packaged in a .sar file, with the following `META-INF/jboss-beans.xml`:

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <!-- This bean is an example of a clustered singleton -->
  <bean name="HASingletonExample" class="org.jboss.ha.examples.HASingletonExample">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>
  </bean>

                                        <bean                    name="ExampleHASingletonController"
 class="org.jboss.ha.singleton.HASingletonController">
    <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>
    <property name="HAPartition"><inject bean="HAPartition"/></property>
    <property name="target"><inject bean="HASingletonExample"/></property>
    <property name="targetStartMethod">startSingleton</property>
    <property name="targetStopMethod">stopSingleton</property>
  </bean>
</deployment>
```

Voila! A clustered singleton service.

The primary advantage of this approach over deploy-ha-singleton. is that the above example can be placed in `deploy` or `farm` and thus can be hot deployed and farmed deployed. Also, if our example service had complex, time-consuming startup requirements, those could potentially be implemented in create() or start() methods. JBoss will invoke create() and start() as soon as the service is deployed; it doesn't wait until the node becomes the master node. So, the service could be primed and ready to go, just waiting for the controller to implement startSingleton() at which point it can immediately provide service.

Although not demonstrated in the example above, the `HASingletonController` can support an optional argument for either or both of the target start and stop methods. These are specified using the `targetStartMethodArgument` and `TargetStopMethodArgument` properties, respectively. Currently, only string values are supported.

4.1.1.3. HASingleton deployments using a Barrier

Services deployed normally inside deploy or farm that want to be started/stopped whenever the content of deploy-hasingleton gets deployed/undeployed, (i.e., whenever the current node becomes the master), need only specify a dependency on the Barrier service:

```
<depends>HASingletonDeployerBarrierController</depends>
```

The way it works is that a BarrierController is deployed along with the HASingletonDeployer and listens for JMX notifications from it. A BarrierController is a relatively simple Mbean that can subscribe to receive any JMX notification in the system. It uses the received notifications to control the lifecycle of a dynamically created Mbean called the Barrier. The Barrier is instantiated, registered and brought to the CREATE state when the BarrierController is deployed. After that, the BarrierController starts and stops the Barrier when matching JMX notifications are received. Thus, other services need only depend on the Barrier bean using the usual <depends> tag, and they will be started and stopped in tandem with the Barrier. When the BarrierController is undeployed the Barrier is also destroyed.

This provides an alternative to the deploy-hasingleton approach in that we can use farming to distribute the service, while content in deploy-hasingleton must be copied manually on all nodes.

On the other hand, the barrier-dependent service will be instantiated/created (i.e., any create() method invoked) on all nodes, but only started on the master node. This is different with the deploy-hasingleton approach that will only deploy (instantiate/create/start) the contents of the deploy-hasingleton directory on one of the nodes.

So services depending on the barrier will need to make sure they do minimal or no work inside their create() step, rather they should use start() to do the work.

> **Note**
>
> The Barrier controls the start/stop of dependent services, but not their destruction, which happens only when the `BarrierController` is itself destroyed/undeployed. Thus using the `Barrier` to control services that need to be "destroyed" as part of their normal “undeploy” operation (like, for example, an `EJBContainer`) will not have the desired effect.

4.1.2. Determining the master node

The various clustered singleton management strategies all depend on the fact that each node in the cluster can independently react to changes in cluster membership and correctly decide whether it is now the “master node”. How is this done?

For each member of the cluster, the HAPartition service maintains an attribute called the CurrentView, which is basically an ordered list of the current members of the cluster. As nodes join and leave the cluster, JGroups ensures that each surviving member of the cluster gets an updated view. You can see the current view by going into the JMX console, and looking at the CurrentView attribute in the `jboss:service=DefaultPartition` mbean. Every member of the cluster will have the same view, with the members in the same order.

Let's say, for example, that we have a 4 node cluster, nodes A through D, and the current view can be expressed as {A, B, C, D}. Generally speaking, the order of nodes in the view will reflect the order in which they joined the cluster (although this is not always the case, and should not be assumed to be the case).

To further our example, let's say there is a singleton service (i.e. an `HASingletonController`) named Foo that's deployed around the cluster, except, for whatever reason, on B. The `HAPartition` service maintains across the cluster a registry of what services are deployed where, in view order. So, on every node in the cluster, the `HAPartition` service knows that the view with respect to the Foo service is {A, C, D} (no B).

Whenever there is a change in the cluster topology of the Foo service, the `HAPartition` service invokes a callback on Foo notifying it of the new topology. So, for example, when Foo started on D, the Foo service running on A, C and D all got callbacks telling them the new view for Foo was {A, C, D}. That callback gives each node enough information to independently decide if it is now the master. The Foo service on each node uses the `HAPartition`'s `HASingletonElectionPolicy` to determine if they are the master, as explained in the *next section*.

If A were to fail or shutdown, Foo on C and D would get a callback with a new view for Foo of {C, D}. C would then become the master. If A restarted, A, C and D would get a callback with a new view for Foo of {C, D, A}. C would remain the master – there's nothing magic about A that would cause it to become the master again just because it was before.

4.1.2.1. HA singleton election policy

The `HASingletonElectionPolicy` object is responsible for electing a master node from a list of available nodes, on behalf of an HA singleton, following a change in cluster topology.

```
public interface HASingletonElectionPolicy
{
  ClusterNode elect(List<ClusterNode> nodes);
}
```

JBoss ships with 2 election policies:

`HASingletonElectionPolicySimple`
: This policy selects a master node based relative age. The desired age is configured via the `position` property, which corresponds to the index in the list of available nodes. `position = 0`, the default, refers to the oldest node; `position = 1`, refers to the 2nd oldest; etc. `position` can also be negative to indicate youngness; imagine the list of available nodes as a circular linked list. `position = -1`, refers to the youngest node; `position = -2`, refers to the 2nd youngest node; etc.

```
<bean class="org.jboss.ha.singleton.HASingletonElectionPolicySimple">
  <property name="position">-1</property>
</bean>
```

`PreferredMasterElectionPolicy`
: This policy extends `HASingletonElectionPolicySimple`, allowing the configuration of a preferred node. The `preferredMaster` property, specified as *host:port* or *address:port*, identifies a specific node that should become master, if available. If the preferred node is not available, the election policy will behave as described above.

```
<bean class="org.jboss.ha.singleton.PreferredMasterElectionPolicy">
  <property name="preferredMaster">server1:12345</property>
</bean>
```

4.2. Farming Deployment

The easiest way to deploy an application into the cluster is to use the farming service. Using the farming service, you can deploy an application (e.g. EAR, WAR, or SAR; either an archive file or in exploded form) to the `all/farm/` directory of any cluster member and the application will be automatically duplicate across all nodes in the same cluster. If a node joins the cluster later, it will

pull in all farm deployed applications in the cluster and deploy them locally at start-up time. If you delete the application from a running clustered server node's `farm/` directory, the application will be undeployed locally and then removed from all other clustered server nodes' `farm/` directories (triggering undeployment).

> **Note**
>
> The farming service was not available in JBoss AS 5.0.0 and 5.0.1. This section is only relevant to releases 5.1.0 and later.

Farming is enabled by default in the `all` configuration in JBoss AS and thus requires no manual setup. The required `farm-deployment-jboss-beans.xml` and `timestamps-jboss-beans.xml` configuration files are located in the `deploy/cluster` directory. If you want to enable farming in a custom configuration, simply copy these files to the corresponding JBoss deploy directory `$JBOSS_HOME/server/your_own_config/deploy/cluster`. Make sure that your custom configuration has clustering enabled.

While there is little need to customize the farming service, it can be customized via the `FarmProfileRepositoryClusteringHandler` bean, whose properties and default values are listed below:

```
<bean name="FarmProfileRepositoryClusteringHandler"
    class="org.jboss.profileservice.cluster.repository.DefaultRepositoryClusteringHandler">

  <property name="partition"><inject bean="HAPartition"/></property>
  <property name="profileDomain">default</property>
  <property name="profileServer">default</property>
  <property name="profileName">farm</property>
  <property name="immutable">false</property>
  <property name="lockTimeout">60000</property><!-- 1 minute -->
  <property name="methodCallTimeout">60000</property><!-- 1 minute -->
  <property name="synchronizationPolicy">
    <inject bean="FarmProfileSynchronizationPolicy"/>
  </property>
</bean>
```

- **partition** is a required attribute to inject the HAPartition service that the farm service uses for intra-cluster communication.
- **profile[Domain|Server|Name]** are all used to identify the profile for which this handler is intended.
- **immutable** indicates whether or not this handler allows a node to push content changes to the cluster. A value of `true` is equivalent to setting `synchronizationPolicy` to

the `org.jboss.system.server.profileservice.repository.clustered.sync` package's `ImmutableSynchronizationPolicy`.

- **lockTimeout** defines the number of milliseconds to wait for cluster-wide lock acquisition.
- **methodCallTimeout** defines the number of milliseconds to wait for invocations on remote cluster nodes.
- **synchronizationPolicy** decides how to handle content additions, reincarnations, updates, or removals from nodes attempting to join the cluster or from cluster merges. The policy is consulted on the "authoritative" node, i.e. the master node for the service on the cluster. *Reincarnation* refers to the phenomenon where a newly started node may contain an application in its `farm` directory that was previously removed by the farming service but might still exist on the starting node if it was not running when the removal took place. The default synchronization policy is defined as follows:

```
<bean name="FarmProfileSynchronizationPolicy"
      class="org.jboss.profileservice.cluster.repository.DefaultSynchronizationPolicy">
  <property name="allowJoinAdditions"><null/></property>
  <property name="allowJoinReincarnations"><null/></property>
  <property name="allowJoinUpdates"><null/></property>
  <property name="allowJoinRemovals"><null/></property>
  <property name="allowMergeAdditions"><null/></property>
  <property name="allowMergeReincarnations"><null/></property>
  <property name="allowMergeUpdates"><null/></property>
  <property name="allowMergeRemovals"><null/></property>
  <property name="developerMode">false</property>
  <property name="removalTrackingTime">2592000000</property><!-- 30 days -->
  <property name="timestampService">
    <inject bean="TimestampDiscrepancyService"/>
  </property>
</bean>
```

- **allow[Join|Merge][Additions|Reincarnations|Updates|Removals]** define fixed responses to requests to allow additions, reincarnations, updates, or removals from joined or merged nodes.
- **developerMode** enables a lenient synchronization policy that allows all changes. Enabling developer mode is equivalent to setting each of the above properties to `true` and is intended for development environments.
- **removalTrackingTime** defines the number of milliseconds for which this policy should remembered removed items, for use in detecting reincarnations.

- **timestampService** estimates and tracks discrepancies in system clocks for current and past members of the cluster. Default implementation is defined in `timestamps-jboss-beans.xml`.

Part II. Clustered Java EE

Clustered JNDI Services

JNDI is one of the most important services provided by the application server. The JBoss HA-JNDI (High Availability JNDI) service brings the following features to JNDI:

- Transparent failover of naming operations. If an HA-JNDI naming Context is connected to the HA-JNDI service on a particular JBoss AS instance, and that service fails or is shut down, the HA-JNDI client can transparently fail over to another AS instance.
- Load balancing of naming operations. An HA-JNDI naming Context will automatically load balance its requests across all the HA-JNDI servers in the cluster.
- Automatic client discovery of HA-JNDI servers (using multicast).
- Unified view of JNDI trees cluster-wide. A client can connect to the HA-JNDI service running on any node in the cluster and find objects bound in JNDI on any other node. This is accomplished via two mechanisms:
  - Cross-cluster lookups. A client can perform a lookup and the server side HA-JNDI service has the ability to find things bound in regular JNDI on any node in the cluster.
  - A replicated cluster-wide context tree. An object bound into the HA-JNDI service will be replicated around the cluster, and a copy of that object will be available in-VM on each node in the cluster.

JNDI is a key component for many other interceptor-based clustering services: those services register themselves with JNDI so the client can look up their proxies and make use of their services. HA-JNDI completes the picture by ensuring that clients have a highly-available means to look up those proxies. However, it is important to understand that using HA-JNDI (or not) has no effect whatsoever on the clustering behavior of the objects that are looked up. To illustrate:

- If an EJB is not configured as clustered, looking up the EJB via HA-JNDI does not somehow result in the addition of clustering capabilities (load balancing of EJB calls, transparent failover, state replication) to the EJB.
- If an EJB is configured as clustered, looking up the EJB via regular JNDI instead of HA-JNDI does not somehow result in the removal of the bean proxy's clustering capabilities.

5.1. How it works

The JBoss client-side HA-JNDI naming Context is based on the client-side interceptor architecture (see *Section 2.2.1, "Client-side interceptor architecture"*). The client obtains an HA-JNDI proxy object (via the `InitialContext` object) and invokes JNDI lookup services on the remote server through the proxy. The client specifies that it wants an HA-JNDI proxy by configuring the naming properties used by the `InitialContext` object. This is covered in detail in *Section 5.2, "Client configuration"*. Other than the need to ensure the appropriate naming properties are provided to

the `InitialContext`, the fact that the naming Context is using HA-JNDI is completely transparent to the client.

On the server side, the HA-JNDI service maintains a cluster-wide context tree. The cluster wide tree is always available as long as there is one node left in the cluster. Each node in the cluster also maintains its own local JNDI context tree. The HA-JNDI service on each node is able to find objects bound into the local JNDI context tree, and is also able to make a cluster-wide RPC to find objects bound in the local tree on any other node. An application can bind its objects to either tree, although in practice most objects are bound into the local JNDI context tree. The design rationale for this architecture is as follows:

- It avoids migration issues with applications that assume that their JNDI implementation is local. This allows clustering to work out-of-the-box with just a few tweaks of configuration files.
- In a homogeneous cluster, this configuration actually cuts down on the amount of network traffic. A homogenous cluster is one where the same types of objects are bound under the same names on each node.
- Designing it in this way makes the HA-JNDI service an optional service since all underlying cluster code uses a straight new `InitialContext` to lookup or create bindings.

On the server side, a naming Context obtained via a call to `new InitialContext()` will be bound to the local-only, non-cluster-wide JNDI Context. So, all EJB homes and such will not be bound to the cluster-wide JNDI Context, but rather, each home will be bound into the local JNDI.

When a remote client does a lookup through HA-JNDI, HA-JNDI will delegate to the local JNDI service when it cannot find the object within the global cluster-wide Context. The detailed lookup rule is as follows.

- If the binding is available in the cluster-wide JNDI tree, return it.
- If the binding is not in the cluster-wide tree, delegate the lookup query to the local JNDI service and return the received answer if available.
- If not available, the HA-JNDI service asks all other nodes in the cluster if their local JNDI service owns such a binding and returns the answer from the set it receives.
- If no local JNDI service owns such a binding, a `NameNotFoundException` is finally raised.

In practice, objects are rarely bound in the cluster-wide JNDI tree; rather they are bound in the local JNDI tree. For example, when EJBs are deployed, their proxies are always bound in local JNDI, not HA-JNDI. So, an EJB home lookup done through HA-JNDI will always be delegated to the local JNDI instance.

Note

If different beans (even of the same type, but participating in different clusters) use the same JNDI name, this means that each JNDI server will have a logically different "target" bound under the same name. (JNDI on node 1 will have a binding

for bean A and JNDI on node 2 will have a binding, under the same name, for bean B). Consequently, if a client performs a HA-JNDI query for this name, the query will be invoked on any JNDI server of the cluster and will return the locally bound stub. Nevertheless, it may not be the correct stub that the client is expecting to receive! So, it is always best practice to ensure that across the cluster different names are used for logically different bindings.

Note

If a binding is only made available on a few nodes in the cluster (for example because a bean is only deployed on a small subset of nodes in the cluster), the probability is higher that a lookup will hit a HA-JNDI server that does not own this binding and thus the lookup will need to be forwarded to all nodes in the cluster. Consequently, the query time will be longer than if the binding would have been available locally. Moral of the story: as much as possible, cache the result of your JNDI queries in your client.

Note

You cannot currently use a non-JNP JNDI implementation (i.e. LDAP) for your local JNDI implementation if you want to use HA-JNDI. However, you can use JNDI federation using the `ExternalContext` MBean to bind non-JBoss JNDI trees into the JBoss JNDI namespace. Furthermore, nothing prevents you using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

5.2. Client configuration

Configuring a client to use HA-JNDI is a matter of ensuring the correct set of naming environment properties are available when a new `InitialContext` is created. How this is done varies depending on whether the client is running inside JBoss AS itself or is in another VM.

5.2.1. For clients running inside the application server

If you want to access HA-JNDI from inside the application server, you must explicitly configure your `InitialContext` by passing in JNDI properties to the constructor. The following code shows how to create a naming Context bound to HA-JNDI:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
// HA-JNDI is listening on the address passed to JBoss via -b
```

```
String bindAddress = System.getProperty("jboss.bind.address", "localhost");
p.put(Context.PROVIDER_URL, bindAddress + ":1100"); // HA-JNDI address and port.
return new InitialContext(p);
```

The Context.PROVIDER_URL property points to the HA-JNDI service configured in the `deploy/cluster/hajndi-jboss-beans.xml` file (see *Section 5.3, "JBoss configuration"*). By default this service listens on the interface named via the `jboss.bind.address` system property, which itself is set to whatever value you assign to the `-b` command line option when you start JBoss AS (or `localhost` if not specified). The above code shows an example of accessing this property.

However, this does not work in all cases, especially when running several JBoss AS instances on the same machine and bound to the same IP address, but configured to use different ports. A safer method is to not specify the Context.PROVIDER_URL but instead allow the `InitialContext` to statically find the in-VM HA-JNDI by specifying the `jnp.partitionName` property:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
// HA-JNDI is registered under the partition name passed to JBoss via -g
String partitionName = System.getProperty("jboss.partition.name", "DefaultPartition");
p.put("jnp.partitionName", partitionName);
return new InitialContext(p);
```

This example uses the `jboss.partition.name` system property to identify the partition with which the HA-JNDI service works. This system property is set to whatever value you assign to the `-g` command line option when you start JBoss AS (or `DefaultPartition` if not specified).

Do not attempt to simplify things by placing a `jndi.properties` file in your deployment or by editing the AS's `conf/jndi.properties` file. Doing either will almost certainly break things for your application and quite possibly across the application server. If you want to externalize your client configuration, one approach is to deploy a properties file not named `jndi.properties`, and then programatically create a `Properties` object that loads that file's contents.

5.2.1.1. Accessing HA-JNDI Resources from EJBs and WARs -- Environment Naming Context

If your HA-JNDI client is an EJB or servlet, the least intrusive way to configure the lookup of resources is to bind the resources to the environment naming context of the bean or webapp performing the lookup. The binding can then be configured to use HA-JNDI instead of a local mapping. Following is an example of doing this for a JMS connection factory and queue (the most common use case for this kind of thing).

Within the bean definition in the ejb-jar.xml or in the war's web.xml you will need to define two resource-ref mappings, one for the connection factory and one for the destination.

```
<resource-ref>
   <res-ref-name>jms/ConnectionFactory</res-ref-name>
   <res-type>javax.jms.QueueConnectionFactory</res-type>
   <res-auth>Container</res-auth>
</resource-ref>

<resource-ref>
   <res-ref-name>jms/Queue</res-ref-name>
   <res-type>javax.jms.Queue</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

Using these examples the bean performing the lookup can obtain the connection factory by looking up 'java:comp/env/jms/ConnectionFactory' and can obtain the queue by looking up 'java:comp/env/jms/Queue'.

Within the JBoss-specific deployment descriptor (jboss.xml for EJBs, jboss-web.xml for a WAR) these references need to be mapped to a URL that makes use of HA-JNDI.

```
<resource-ref>
   <res-ref-name>jms/ConnectionFactory</res-ref-name>
   <jndi-name>jnp://${jboss.bind.address}:1100/ConnectionFactory</jndi-name>
</resource-ref>

<resource-ref>
   <res-ref-name>jms/Queue</res-ref-name>
   <jndi-name>jnp://${jboss.bind.address}:1100/queue/A</jndi-name>
 </resource-ref>
```

The URL should be the URL to the HA-JNDI server running on the same node as the bean; if the bean is available the local HA-JNDI server should also be available. The lookup will then automatically query all of the nodes in the cluster to identify which node has the JMS resources available.

The `${jboss.bind.address}` syntax used above tells JBoss to use the value of the `jboss.bind.address` system property when determining the URL. That system property is itself set to whatever value you assign to the `-b` command line option when you start JBoss AS.

5.2.1.2. Why do this programmatically and not just put this in a jndi.properties file?

The JBoss application server's internal naming environment is controlled by the `conf/jndi.properties` file, which should not be edited.

No other jndi.properties file should be deployed inside the application server because of the possibility of its being found on the classpath when it shouldn't and thus disrupting the internal operation of the server. For example, if an EJB deployment included a jndi.properties configured for HA-JNDI, when the server binds the EJB proxies into JNDI it will likely bind them into the replicated HA-JNDI tree and not into the local JNDI tree where they belong.

5.2.1.3. How can I tell if things are being bound into HA-JNDI that shouldn't be?

Go into the the jmx-console and execute the `list` operation on the `jboss:service=JNDIView` mbean. Towards the bottom of the results, the contents of the "HA-JNDI Namespace" are listed. Typically this will be empty; if any of your own deployments are shown there and you didn't explicitly bind them there, there's probably an improper jndi.properties file on the classpath. Please visit the following link for an example: *Problem with removing a Node from Cluster* [http://www.jboss.com/index.html?module=bb&op=viewtopic&t=104715]

5.2.2. For clients running outside the application server

The JNDI client needs to be aware of the HA-JNDI cluster. You can pass a list of JNDI servers (i.e., the nodes in the HA-JNDI cluster) to the `java.naming.provider.url` JNDI setting in the `jndi.properties` file. Each server node is identified by its IP address and the JNDI port number. The server nodes are separated by commas (see *Section 5.3, "JBoss configuration"* for how to configure the servers and ports).

```
java.naming.provider.url=server1:1100,server2:1100,server3:1100,server4:1100
```

When initialising, the JNP client code will try to get in touch with each server node from the list, one after the other, stopping as soon as one server has been reached. It will then download the HA-JNDI stub from this node.

> **Note**
>
> There is no load balancing behavior in the JNP client lookup process itself. It just goes through the provider lists and uses the first available server to obtain the stub. The HA-JNDI provider list only needs to contain a subset of HA-JNDI nodes in the cluster; once the HA-JNDI stub is downloaded, the stub will include information on all the available servers. A good practice is to include a set of servers such that you are certain that at least one of those in the list will be available.

The downloaded smart proxy contains the list of currently running nodes and the logic to load balance naming requests and to fail-over to another node if necessary. Furthermore, each time a JNDI invocation is made to the server, the list of targets in the proxy interceptor is updated (only if the list has changed since the last call).

If the property string `java.naming.provider.url` is empty or if all servers it mentions are not reachable, the JNP client will try to discover a HA-JNDI server through a multicast call on the network (auto-discovery). See *Section 5.3, "JBoss configuration"* for how to configure auto-discovery on the JNDI server nodes. Through auto-discovery, the client might be able to get a valid HA-JNDI server node without any configuration. Of course, for auto-discovery to work, the network segment(s) between the client and the server cluster must be configured to propagate such multicast datagrams.

> **Note**
>
> By default the auto-discovery feature uses multicast group address 230.0.0.4 and port 1102.

In addition to the `java.naming.provider.url` property, you can specify a set of other properties. The following list shows all clustering-related client side properties you can specify when creating a new `InitialContext`. (All of the standard, non-clustering-related environment properties used with regular JNDI are also available.)

- `java.naming.provider.url`: Provides a list of IP addresses and port numbers for HA-JNDI provider nodes in the cluster. The client tries those providers one by one and uses the first one that responds.
- `jnp.disableDiscovery`: When set to `true`, this property disables the automatic discovery feature. Default is `false`.
- `jnp.partitionName`: In an environment where multiple HA-JNDI services bound to distinct clusters (a.k.a. partitions), are running, this property allows you to ensure that your client only accepts automatic-discovery responses from servers in the desired partition. If you do not use the automatic discovery feature (i.e. jnp.disableDiscovery is true), this property is not used. By default, this property is not set and the automatic discovery selects the first HA-JNDI server that responds, regardless of the cluster partition name.
- `jnp.discoveryTimeout`: Determines how many milliseconds the context will wait for a response to its automatic discovery packet. Default is 5000 ms.
- `jnp.discoveryGroup`: Determines which multicast group address is used for the automatic discovery. Default is 230.0.0.4. Must match the value of the AutoDiscoveryAddress configured on the server side HA-JNDI service. Note that the server side HA-JNDI service by default listens on the address specified via the `-u` startup switch, so if `-u` is used on the server side (as is recommended), jnp.discoveryGroup will need to be configured on the client side.
- `jnp.discoveryPort`: Determines which multicast port is used for the automatic discovery. Default is 1102. Must match the value of the AutoDiscoveryPort configured on the server side HA-JNDI service.
- `jnp.discoveryTTL`: specifies the TTL (time-to-live) f or autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate

before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.

Since AS 5.1.0.GA, there's a new system property called `jboss.global.jnp.disableDiscovery` that controls autodiscovery behaivour at the client VM level and it can take `true` or `false` (default value) boolean values. The system property works in such way that if missing or it's set to `false`, default auto discovery behaivour will be used. If set to `true`, the following can happen:

- If jnp.DisableDiscovery not present when context is created, auto discovery will not be attempted.
- If jnp.DisableDiscovery is present when context is created and this is set to true, auto discovery will not be attempted.
- If jnp.DisableDiscovery is present when context is created and is set to false, the global auto discovery system property is ignored and auto discovery will be attempted. This enables global discovery client VM setting to be overriden.

5.3. JBoss configuration

The `hajndi-jboss-beans.xml` file in the `JBOSS_HOME/server/all/deploy/cluster` directory includes the following bean to enable HA-JNDI services.

```
<bean name="HAJNDI" class="org.jboss.ha.jndi.HANamingService">

 <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>

 <!-- The partition used for group RPCs to find locally bound objects on other nodes -->
 <property name="HAPartition"><inject bean="HAPartition"/></property>

 <!-- Handler for the replicated tree -->
 <property name="distributedTreeManager">
  <bean class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
   <property name="cacheHandler"><inject bean="HAPartitionCacheHandler"/></property>
  </bean>
 </property>

 <property name="localNamingInstance">
  <inject bean="jboss:service=NamingBeanImpl" property="namingInstance"/>
 </property>

 <!-- The thread pool used to control the bootstrap and auto discovery lookups -->
 <property name="lookupPool"><inject bean="jboss.system:service=ThreadPool"/></property>

 <!-- Bind address of bootstrap endpoint -->
 <property name="bindAddress">${jboss.bind.address}</property>
```

```
<!-- Port on which the HA-JNDI stub is made available -->
<property name="port">
  <!-- Get the port from the ServiceBindingManager -->
  <value-factory bean="ServiceBindingManager" method="getIntBinding">
    <parameter>jboss:service=HAJNDI</parameter>
    <parameter>Port</parameter>
  </value-factory>
</property>

<!-- Bind address of the HA-JNDI RMI endpoint -->
<property name="rmiBindAddress">${jboss.bind.address}</property>

<!-- RmiPort to be used by the HA-JNDI service once bound. 0 = ephemeral. -->
<property name="rmiPort">
  <!-- Get the port from the ServiceBindingManager -->
  <value-factory bean="ServiceBindingManager" method="getIntBinding">
    <parameter>jboss:service=HAJNDI</parameter>
    <parameter>RmiPort</parameter>
  </value-factory>
</property>

<!-- Accept backlog of the bootstrap socket -->
<property name="backlog">50</property>

<!-- A flag to disable the auto discovery via multicast -->
<property name="discoveryDisabled">false</property>
<!-- Set the auto-discovery bootstrap multicast bind address. If not
       specified and a BindAddress is specified, the BindAddress will be used. -->
<property name="autoDiscoveryBindAddress">${jboss.bind.address}</property>
<!-- Multicast Address and group port used for auto-discovery -->
<property name="autoDiscoveryAddress">${jboss.partition.udpGroup:230.0.0.4}</property>
<property name="autoDiscoveryGroup">1102</property>
<!-- The TTL (time-to-live) for autodiscovery IP multicast packets -->
<property name="autoDiscoveryTTL">16</property>

<!-- The load balancing policy for HA-JNDI -->
<property name="loadBalancePolicy">
    org.jboss.ha.framework.interfaces.RoundRobin
</property>

<!-- Client socket factory to be used for client-server
       RMI invocations during JNDI queries
<property name="clientSocketFactory">custom</property>
-->
```

```
  <!-- Server socket factory to be used for client-server
        RMI invocations during JNDI queries
  <property name="serverSocketFactory">custom</property>
  -->
</bean>
```

You can see that this bean has a number of other services injected into different properties:

- **HAPartition** accepts the core clustering service used manage HA-JNDI's clustered proxies and to make the group RPCs that find locally bound objects on other nodes. See *Section 3.3, "The HAPartition Service"* for more.
- **distributedTreeManager** accepts a handler for the replicated tree. The standard handler uses JBoss Cache to manage the replicated tree. The JBoss Cache instance is retrieved using the injected `HAPartitionCacheHandler` bean. See *Section 3.3, "The HAPartition Service"* for more details.
- **localNamingInstance** accepts the reference to the local JNDI service.
- **lookupPool** accepts the thread pool used to provide threads to handle the bootstrap and auto discovery lookups.

Besides the above dependency injected services, the available configuration attributes for the HAJNDI bean are as follows:

- **bindAddress** specifies the address to which the HA-JNDI server will bind to listen for naming proxy download requests from JNP clients. The default value is the value of the `jboss.bind.address` system property, or `localhost` if that property is not set. The `jboss.bind.address` system property is set if the `-b` command line switch is used when JBoss is started.
- **port** specifies the port to which the HA-JNDI server will bind to listen for naming proxy download requests from JNP clients. The value is obtained from the ServiceBindingManager bean configured in `conf/bootstrap/bindings.xml`. The default value is `1100`.
- **Backlog** specifies the maximum queue length for incoming connection indications for the TCP server socket on which the service listens for naming proxy download requests from JNP clients. The default value is `50`.
- **rmiBindAddress** specifies the address to which the HA-JNDI server will bind to listen for RMI requests (e.g. for JNDI lookups) from naming proxies. The default value is the value of the `jboss.bind.address` system property, or `localhost` if that property is not set. The `jboss.bind.address` system property is set if the `-b` command line switch is used when JBoss is started.
- **rmiPort** specifies the port to which the server will bind to communicate with the downloaded stub. The value is obtained from the ServiceBindingManager bean configured in `conf/`

`bootstrap/bindings.xml`. The default value is `1101`. If no value is set, the operating system automatically assigns a port.

- **discoveryDisabled** is a boolean flag that disables configuration of the auto discovery multicast listener. The default is `false`.
- **autoDiscoveryAddress** specifies the multicast address to listen to for JNDI automatic discovery. The default value is the value of the `jboss.partition.udpGroup` system property, or 230.0.0.4 if that is not set. The `jboss.partition.udpGroup` system property is set if the `-u` command line switch is used when JBoss is started.
- **autoDiscoveryGroup** specifies the port to listen on for multicast JNDI automatic discovery packets. The default value is `1102`.
- **autoDiscoveryBindAddress** sets the interface on which HA-JNDI should listen for auto-discovery request packets. If this attribute is not specified and a `bindAddress` is specified, the `bindAddress` will be used.
- **autoDiscoveryTTL** specifies the TTL (time-to-live) for autodiscovery IP multicast packets. This value represents the number of network hops a multicast packet can be allowed to propagate before networking equipment should drop the packet. Despite its name, it does not represent a unit of time.
- **loadBalancePolicy** specifies the class name of the LoadBalancePolicyimplementation that should be included in the client proxy. See *???* for details.
- **clientSocketFactory** is an optional attribute that specifies the fully qualified classname of the `java.rmi.server.RMIClientSocketFactory` that should be used to create client sockets. The default is `null`.
- **serverSocketFactory** is an optional attribute that specifies the fully qualified classname of the `java.rmi.server.RMIServerSocketFactory` that should be used to create server sockets. The default is `null`.

5.3.1. Adding a Second HA-JNDI Service

It is possible to start several HA-JNDI services that use different HAPartitions. This can be used, for example, if a node is part of many logical clusters. In this case, make sure that you set a different port or IP address for each service. For instance, if you wanted to hook up HA-JNDI to the example cluster you set up and change the binding port, the bean descriptor would look as follows (properties that do not vary from the standard deployments are omitted):

```
<-- Cache Handler for secondary HAPartition -->
<bean name="SecondaryHAPartitionCacheHandler"
      class="org.jboss.ha.framework.server.HAPartitionCacheHandlerImpl">
  <property name="cacheManager"><inject bean="CacheManager"/></property>
  <property name="cacheConfigName">secondary-ha-partition</property>
```

```
</bean>

<-- The secondary HAPartition -->
<bean name="SecondaryHAPartition" class="org.jboss.ha.framework.server.ClusterPartition">
  <depends>jboss:service=Naming</depends>

  <property name="cacheHandler">
    <inject bean="SecondaryHAPartitionCacheHandler"/>
  </property>
  <property name="partitionName">SecondaryPartition</property>

  ....
</bean>

<bean name="MySpecialPartitionHAJNDI" class="org.jboss.ha.jndi.HANamingService">
  <property name="HAPartition"><inject bean="SecondaryHAPartition"/></property>

  <property name="distributedTreeManager">
    <bean class="org.jboss.ha.jndi.impl.jbc.JBossCacheDistributedTreeManager">
      <property name="cacheHandler">
        <inject bean="SecondaryHAPartitionPartitionCacheHandler"/>
      </property>
    </bean>
  </property>

  <property name="port">56789</property>
  <property name="rmiPort">56790</property>
  <property name="autoDiscoveryGroup">56791</property>

  .....
</bean>
```

Clustered Session EJBs

Session EJBs provide remote invocation services. They are clustered based on the client-side interceptor architecture. The client application for a clustered session bean is the same as the client for the non-clustered version of the session bean, except for some minor changes. No code change or re-compilation is needed on the client side. Now, let's check out how to configure clustered session beans in EJB 3.0 and EJB 2.x server applications respectively.

6.1. Stateless Session Bean in EJB 3.0

Clustering stateless session beans is probably the easiest case since no state is involved. Calls can be load balanced to any participating node (i.e. any node that has this specific bean deployed) of the cluster.

To cluster a stateless session bean in EJB 3.0, simply annotate the bean class with the `@Clustered` annotation. This annotation contains optional parameters for overriding both the load balance policy and partition to use.

```
public @interface Clustered
{
   String partition() default "${jboss.partition.name:DefaultPartition}";
   String loadBalancePolicy() default "LoadBalancePolicy";
}
```

- **partition** specifies the name of the cluster the bean participates in. While the `@Clustered` annotation lets you override the default partition, `DefaultPartition`, for an individual bean, you can override this for all beans using the `jboss.partition.name` system property.
- **loadBalancePolicy** defines the name of a class implementing `org.jboss.ha.client.loadbalance.LoadBalancePolicy`, indicating how the bean stub should balance calls made on the nodes of the cluster. The default value, `LoadBalancePolicy` is a special token indicating the default policy for the session bean type. For stateless session beans, the default policy is `org.jboss.ha.client.loadbalance.RoundRobin`. You can override the default value using your own implementation, or choose one from the list of available policies:

  `org.jboss.ha.client.loadbalance.RoundRobin`
  : Starting with a random target, always favors the next available target in the list, ensuring maximum load balancing always occurs.

  `org.jboss.ha.client.loadbalance.RandomRobin`
  : Randomly selects its target without any consideration to previously selected targets.

`org.jboss.ha.client.loadbalance.aop.FirstAvailable`
: Once a target is chosen, always favors that same target; i.e. no further load balancing occurs. Useful in cases where "sticky session" behavior is desired, e.g. stateful session beans.

`org.jboss.ha.client.loadbalance.aop.FirstAvailableIdenticalAllProxies`
: Similar to `FirstAvailable`, except that the favored target is shared across all proxies.

Here is an example of a clustered EJB 3.0 stateless session bean implementation.

```
@Stateless
@Clustered
public class MyBean implements MySessionInt
{
  public void test()
  {
    // Do something cool
  }
}
```

Rather than using the `@Clustered` annotation, you can also enable clustering for a session bean in jboss.xml:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cluster-config>
        <partition-name>FooPartition</partition-name>
        <load-balance-policy>
          org.jboss.ha.framework.interfaces.RandomRobin
        </load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

> **Note**
>
> The `<clustered>true</clustered>` element is really just an alias for the `<container-name>Clustered Stateless SessionBean</container-name>` element in the conf/standardjboss.xml file.

In the bean configuration, only the <clustered> element is necessary to indicate that the bean needs to support clustering features. The default values for the optional <cluster-config> elements match those of the corresponding properties from the `@Clustered` annotation.

6.2. Stateful Session Beans in EJB 3.0

Clustering stateful session beans is more complex than clustering their stateless counterparts since JBoss needs to manage the state information. The state of all stateful session beans are replicated and synchronized across the cluster each time the state of a bean changes.

6.2.1. The EJB application configuration

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the `@Clustered` annotation, just as we did with the EJB 3.0 stateless session bean earlier. In contrast to stateless session beans, stateful session bean method invocations are load balanced using `org.jboss.ha.client.loadbalance.aop.FirstAvailable` policy, by default. Using this policy, methods invocations will stick to a randomly chosen node.

The `@org.jboss.ejb3.annotation.CacheConfig` annotation can also be applied to the bean to override the default caching behavior. Below is the definition of the `@CacheConfig` annotation:

```
public @interface CacheConfig
{
   String name() default "";
   int maxSize() default 10000;
   long idleTimeoutSeconds() default 300;
   boolean replicationIsPassivation() default true;
   long removalTimeoutSeconds() default 0;
}
```

- `name` specifies the name of a cache configuration registered with the `CacheManager` service discussed in *Section 6.2.3, "CacheManager service configuration"*. By default, the `sfsb-cache` configuration will be used.
- `maxSize` specifies the maximum number of beans that can cached before the cache should start passivating beans, using an LRU algorithm.

- `idleTimeoutSeconds` specifies the max period of time a bean can go unused before the cache should passivate it (irregardless of whether maxSize beans are cached.)
- `removalTimeoutSeconds` specifies the max period of time a bean can go unused before the cache should remove it altogether.
- `replicationIsPassivation` specifies whether the cache should consider a replication as being equivalent to a passivation, and invoke any @PrePassivate and @PostActivate callbacks on the bean. By default true, since replication involves serializing the bean, and preparing for and recovering from serialization is a common reason for implementing the callback methods.

Here is an example of a clustered EJB 3.0 stateful session bean implementation.

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000, removalTimeoutSeconds=18000)
public class MyBean implements MySessionInt
{
  private int state = 0;

  public void increment()
  {
    System.out.println("counter: " + (state++));
  }
}
```

As with stateless beans, the @Clustered annotation can alternatively be omitted and the clustering configuration instead applied to jboss.xml:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>NonAnnotationStateful</ejb-name>
      <clustered>true</clustered>
      <cache-config>
        <cache-max-size>5000</cache-max-size>
        <remove-timeout-seconds>18000</remove-timeout-seconds>
      </cache-config>
    </session>
  </enterprise-beans>
</jboss>
```

6.2.2. Optimize state replication

As the replication process is a costly operation, you can optimise this behaviour by optionally implementing the org.jboss.ejb3.cache.Optimized interface in your bean class:

```
public interface Optimized
{
   boolean isModified();
}
```

Before replicating your bean, the container will check if your bean implements the `Optimized` interface. If this is the case, the container calls the `isModified()` method and will only replicate 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 would not occur.

6.2.3. CacheManager service configuration

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The `CacheManager` service, described in *Section 3.2.1, "The JBoss AS CacheManager Service"* is both a factory and registry of JBoss Cache instances. By default, stateful session beans use the `sfsb-cache` configuration from the `CacheManager`, defined as follows:

```
<bean name="StandardSFSBCacheConfig" class="org.jboss.cache.config.Configuration">

  <!--  No transaction manager lookup -->

  <!-- Name of cluster. Needs to be the same for all members -->
  <property name="clusterName">
    ${jboss.partition.name:DefaultPartition}-SFSBCache
  </property>
  <!--
    Use a UDP (multicast) based stack. Need JGroups flow control (FC)
    because we are using asynchronous replication.
  -->
  <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
  <property name="fetchInMemoryState">true</property>

  <property name="nodeLockingScheme">PESSIMISTIC</property>
  <property name="isolationLevel">REPEATABLE_READ</property>
  <property name="useLockStriping">false</property>
  <property name="cacheMode">REPL_ASYNC</property>
```

```
<!--
  Number of milliseconds to wait until all responses for a
  synchronous call have been received. Make this longer
  than lockAcquisitionTimeout.
-->
<property name="syncReplTimeout">17500</property>
<!-- Max number of milliseconds to wait for a lock acquisition -->
<property name="lockAcquisitionTimeout">15000</property>
<!-- The max amount of time (in milliseconds) we wait until the
state (ie. the contents of the cache) are retrieved from
existing members at startup. -->
<property name="stateRetrievalTimeout">60000</property>

<!--
  SFSBs use region-based marshalling to provide for partial state
  transfer during deployment/undeployment.
-->
<property name="useRegionBasedMarshalling">false</property>
<!-- Must match the value of "useRegionBasedMarshalling" -->
<property name="inactiveOnStartup">false</property>

<!-- Disable asynchronous RPC marshalling/sending -->
<property name="serializationExecutorPoolSize">0</property>
<!-- We have no asynchronous notification listeners -->
<property name="listenerAsyncPoolSize">0</property>

<property name="exposeManagementStatistics">true</property>

<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">

    <!--  Just set to true to turn on buddy replication -->
    <property name="enabled">false</property>

    <!--
      A way to specify a preferred replication group.  We try
      and pick a buddy who shares the same pool name (falling
      back to other buddies if not available).
    -->
    <property name="buddyPoolName">default</property>

    <property name="buddyCommunicationTimeout">17500</property>

    <!-- Do not change these -->
```

```
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>

    <property name="buddyLocatorConfig">
      <bean class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
        <!-- The number of backup nodes we maintain -->
        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
             a different physical host. If not able to do so
             though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>
      </bean>
    </property>
  </bean>
</property>
<property name="cacheLoaderConfig">
  <bean class="org.jboss.cache.config.CacheLoaderConfig">
    <!-- Do not change these -->
    <property name="passivation">true</property>
    <property name="shared">false</property>

    <property name="individualCacheLoaderConfigs">
      <list>
        <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
          <!-- Where passivated sessions are stored -->
          <property name="location">${jboss.server.data.dir}${/}sfsb</property>
          <!-- Do not change these -->
          <property name="async">false</property>
          <property name="fetchPersistentState">true</property>
          <property name="purgeOnStartup">true</property>
          <property name="ignoreModifications">false</property>
          <property name="checkCharacterPortability">false</property>
        </bean>
      </list>
    </property>
  </bean>
</property>

<!-- EJBs use JBoss Cache eviction -->
<property name="evictionConfig">
  <bean class="org.jboss.cache.config.EvictionConfig">
    <property name="wakeupInterval">5000</property>
    <!--  Overall default -->
```

```
    <property name="defaultEvictionRegionConfig">
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
        </property>
      </bean>
    </property>
    <!-- EJB3 integration code will programatically create other regions as beans are deployed -->
  </bean>
 </property>
</bean>
```

Eviction

The default SFSB cache is configured to support eviction. The EJB3 SFSB container uses the JBoss Cache eviction mechanism to manage SFSB passivation. When beans are deployed, the EJB container will programatically add eviction regions to the cache, one region per bean type.

CacheLoader

A JBoss Cache CacheLoader is also configured; again to support SFSB passivation. When beans are evicted from the cache, the cache loader passivates them to a persistent store; in this case to the filesystem in the $JBOSS_HOME/server/all/data/sfsb directory. JBoss Cache supports a variety of different CacheLoader implementations that know how to store data to different persistent store types; see the JBoss Cache documentation for details. However, if you change the CacheLoaderConfiguration, be sure that you do not use a shared store, e.g. a single schema in a shared database. Each node in the cluster must have its own persistent store, otherwise as nodes independently passivate and activate clustered beans, they will corrupt each other's data.

Buddy Replication

Using buddy replication, state is replicated to a configurable number of backup servers in the cluster (aka buddies), rather than to all servers in the cluster. To enable buddy replication, adjust the following properties in the `buddyReplicationConfig` property bean:

- Set `enabled` to `true`.
- Use the `buddyPoolName` to form logical subgroups of nodes within the cluster. If possible, buddies will be chosen from nodes in the same buddy pool.
- Adjust the `buddyLocatorConfig.numBuddies` property to reflect the number of backup nodes to which each node should replicate its state.

6.3. Stateless Session Bean in EJB 2.x

To make an EJB 2.x bean clustered, you need to modify its `jboss.xml` descriptor to contain a `<clustered>` tag.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>nextgen.StatelessSession</ejb-name>
      <jndi-name>nextgen.StatelessSession</jndi-name>
      <clustered>true</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </home-load-balance-policy>
        <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </bean-load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

- **partition-name** specifies the name of the cluster the bean participates in. The default value is `DefaultPartition`. The default partition name can also be set system-wide using the `jboss.partition.name` system property.
- **home-load-balance-policy** indicates the class to be used by the home stub to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a `RoundRobin` fashion.
- **bean-load-balance-policy** Indicates the class to be used by the bean stub to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a `RoundRobin` fashion.

6.4. Stateful Session Bean in EJB 2.x

Clustering stateful session beans is more complex than clustering their stateless counterparts since JBoss needs to manage the state information. The state of all stateful session beans are replicated and synchronized across the cluster each time the state of a bean changes. The JBoss AS uses the `HASessionStateService` bean to manage distributed session states for clustered EJB 2.x stateful session beans. In this section, we cover both the session bean configuration and the `HASessionStateService` bean configuration.

6.4.1. The EJB application configuration

In the EJB application, you need to modify the `jboss.xml` descriptor file for each stateful session bean and add the `<clustered>` tag.

```
<jboss>
 <enterprise-beans>
  <session>
   <ejb-name>nextgen.StatefulSession</ejb-name>
   <jndi-name>nextgen.StatefulSession</jndi-name>
   <clustered>True</clustered>
   <cluster-config>
    <partition-name>DefaultPartition</partition-nam>
    <home-load-balance-policy>
     org.jboss.ha.framework.interfaces.RoundRobin
    </home-load-balance-policy>
    <bean-load-balance-policy>
     org.jboss.ha.framework.interfaces.FirstAvailable
    </bean-load-balance-policy>
    <session-state-manager-jndi-name>
     /HASessionState/Default
    </session-state-manager-jndi-name>
   </cluster-config>
  </session>
 </enterprise-beans>
</jboss>
```

In the bean configuration, only the `<clustered>` tag is mandatory to indicate that the bean works in a cluster. The `<cluster-config>` element is optional and its default attribute values are indicated in the sample configuration above.

The `<session-state-manager-jndi-name>` tag is used to give the JNDI name of the `HASessionStateService` to be used by this bean.

The description of the remaining tags is identical to the one for stateless session bean. Actions on the clustered stateful session bean's home interface are by default load-balanced, round-robin. Once the bean's remote stub is available to the client, calls will not be load-balanced round-robin any more and will stay "sticky" to the first node in the list.

6.4.2. Optimize state replication

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

```
public boolean isModified();
```

Before replicating your bean, the container will detect if your bean implements this method. 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 would not occur. This feature is available on JBoss AS 3.0.1+ only.

6.4.3. The HASessionStateService configuration

The `HASessionStateService` bean is defined in the `all/deploy/cluster/ha-legacy-jboss-beans.xml` file.

```
<bean name="HASessionStateService"
      class="org.jboss.ha.hasessionstate.server.HASessionStateService">

  <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(...)</annotation>

  <!--  Partition used for group RPCs -->
  <property name="HAPartition"><inject bean="HAPartition"/></property>

  <!-- JNDI name under which the service is bound -->
  <property name="jndiName">/HASessionState/Default</property>
  <!-- Max delay before cleaning unreclaimed state.
       Defaults to 30*60*1000 => 30 minutes -->
  <property name="beanCleaningDelay">0</property>

</bean>
```

The configuration attributes in the `HASessionStateService` bean are listed below.

- **HAPartition** is a required attribute to inject the HAPartition service that HA-JNDI uses for intra-cluster communication.
- **jndiName** is an optional attribute to specify the JNDI name under which this `HASessionStateService` bean is bound. The default value is `/HAPartition/Default`.
- **beanCleaningDelay** is an optional attribute to specify the number of miliseconds after which the `HASessionStateService` can clean a state that has not been modified. If a node, owning a bean, crashes, its brother node will take ownership of this bean. Nevertheless, the container cache of the brother node will not know about it (because it has never seen it before) and will never delete according to the cleaning settings of the bean. That is why the

`HASessionStateService` needs to do this cleanup sometimes. The default value is `30*60*1000` milliseconds (i.e., 30 minutes).

6.4.4. Handling Cluster Restart

We have covered the HA smart client architecture in the section called “Client-side interceptor architecture”. The default HA smart proxy client can only failover as long as one node in the cluster exists. If there is a complete cluster shutdown, the proxy becomes orphaned and loses knowledge of the available nodes in the cluster. There is no way for the proxy to recover from this. The proxy needs to look up a fresh set of targets out of JNDI/HAJNDI when the nodes are restarted.

The 3.2.7+/4.0.2+ releases contain a RetryInterceptor that can be added to the proxy client side interceptor stack to allow for a transparent recovery from such a restart failure. To enable it for an EJB, setup an invoker-proxy-binding that includes the RetryInterceptor. Below is an example jboss.xml configuration.

```
<jboss>
 <session>
  <ejb-name>nextgen_RetryInterceptorStatelessSession</ejb-name>
  <invoker-bindings>
   <invoker>
    <invoker-proxy-binding-name>
      clustered-retry-stateless-rmi-invoker
    </invoker-proxy-binding-name>
    <jndi-name>nextgen_RetryInterceptorStatelessSession</jndi-name>
   </invoker>
  </invoker-bindings>
  <clustered>true</clustered>
 </session>
 <invoker-proxy-binding>
  <name>clustered-retry-stateless-rmi-invoker</name>
  <invoker-mbean>jboss:service=invoker,type=jrmpha</invoker-mbean>
  <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
  <proxy-factory-config>
   <client-interceptors>
    <home>
     <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
     <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
     <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
     <interceptor>org.jboss.proxy.ejb.RetryInterceptor</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.proxy.ejb.RetryInterceptor</interceptor>
        <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
      </bean>
    </client-interceptors>
  </proxy-factory-config>
 </invoker-proxy-binding>
</jboss>
```

6.4.5. JNDI Lookup Process

In order to recover the HA proxy, the RetryInterceptor does a lookup in JNDI. This means that internally it creates a new InitialContext and does a JNDI lookup. But, for that lookup to succeed, the InitialContext needs to be configured properly to find your naming server. The RetryInterceptor will go through the following steps in attempting to determine the proper naming environment properties:

1. It will check its own static retryEnv field. This field can be set by client code via a call to RetryInterceptor.setRetryEnv(Properties). This approach to configuration has two downsides: first, it reduces portability by introducing JBoss-specific calls to the client code; and second, since a static field is used only a single configuration per JVM is possible.
2. If the retryEnv field is null, it will check for any environment properties bound to a ThreadLocal by the org.jboss.naming.NamingContextFactory class. To use this class as your naming context factory, in your jndi.properties set property java.naming.factory.initial=org.jboss.naming.NamingContextFactory. The advantage of this approach is use of org.jboss.naming.NamingContextFactory is simply a configuration option in your jndi.properties file, and thus your java code is unaffected. The downside is the naming properties are stored in a ThreadLocal and thus are only visible to the thread that originally created an InitialContext.
3. If neither of the above approaches yield a set of naming environment properties, a default InitialContext is used. If the attempt to contact a naming server is unsuccessful, by default the InitialContext will attempt to fall back on multicast discovery to find an HA-JNDI naming server. See the section on “ClusteredJNDI Services” for more on multicast discovery of HA-JNDI.

6.4.6. SingleRetryInterceptor

The RetryInterceptor is useful in many use cases, but a disadvantage it has is that it will continue attempting to re-lookup the HA proxy in JNDI until it succeeds. If for some reason it cannot succeed, this process could go on forever, and thus the EJB call that triggered the RetryInterceptor will never return. For many client applications, this possibility is unacceptable. As a result, JBoss doesn't make the RetryInterceptor part of its default client interceptor stacks for clustered EJBs.

In the 4.0.4.RC1 release, a new flavor of retry interceptor was introduced, the org.jboss.proxy.ejb.SingleRetryInterceptor. This version works like the RetryInterceptor, but only

makes a single attempt to re-lookup the HA proxy in JNDI. If this attempt fails, the EJB call will fail just as if no retry interceptor was used. Beginning with 4.0.4.CR2, the SingleRetryInterceptor is part of the default client interceptor stacks for clustered EJBs.

The downside of the SingleRetryInterceptor is that if the retry attempt is made during a portion of a cluster restart where no servers are available, the retry will fail and no further attempts will be made.

Clustered Entity EJBs

In a JBoss AS cluster, entity bean instance caches need to be kept in sync across all nodes. If an entity bean provides remote services, the service methods need to be load balanced as well.

7.1. Entity Bean in EJB 3.0

In EJB 3.0, entity beans primarily serve as a persistence data model. They do not provide remote services. Hence, the entity bean clustering service in EJB 3.0 primarily deals with distributed caching and replication, instead of load balancing.

7.1.1. Configure the distributed cache

To avoid round trips to the database, you can use a cache for your entities. JBoss EJB 3.0 entity beans are implemented by Hibernate, which has support for a second-level cache. The second-level cache provides the following functionalities:

- If you persist a cache-enabled entity bean instance to the database via the entity manager, the entity will be inserted into the cache.
- If you update an entity bean instance, and save the changes to the database via the entity manager, the entity will be updated in the cache.
- If you remove an entity bean instance from the database via the entity manager, the entity will be removed from the cache.
- If loading a cached entity from the database via the entity manager, and that entity does not exist in the database, it will be inserted into the cache.

As well as a region for caching entities, the second-level cache also contains regions for caching collections, queries, and timestamps. The Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying second-level cache implementation.

Configuration of a the second-level cache is done via your EJB3 deployment's persistence.xml.

e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/persistence"
  <persistence-unit name="tempdb" transaction-type="JTA">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.use_second_level_cache" value="true"/>
      <property name="hibernate.cache.use_query_cache" value="true"/>
```

```
    <property name="hibernate.cache.region.factory_class"
      value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
    <!-- region factory specific properties -->
    <property name="hibernate.cache.region.jbc2.cachefactory" value="java:CacheManager"/>
    <property name="hibernate.cache.region.jbc2.cfg.entity" value="mvcc-entity"/>
    <property name="hibernate.cache.region.jbc2.cfg.collection" value="mvcc-entity"/>
   </properties>
  </persistence-unit>
</persistence>
```

hibernate.cache.use_second_level_cache
Enables second-level caching of entities and collections.

hibernate.cache.use_query_cache
Enables second-level caching of queries.

hibernate.cache.region.jbc2.query.localonly
If you have enabled caching of query results, set to `true` to tell Hibernate you want to suppress costly replication of those results around the cluster. No need to set this property if you want query results replicated.

hibernate.cache.region.factory_class
Defines the `RegionFactory` implementation that dictates region-specific caching behavior. Hibernate ships with 2 types of JBoss Cache-based second-level caches: shared and multiplexed.

A shared region factory uses the same Cache for all cache regions - much like the legacy CacheProvider implementation in older Hibernate versions.

Hibernate ships with 2 shared region factory implementations:

org.hibernate.cache.jbc2.SharedJBossCacheRegionFactory
Uses a single JBoss Cache configuration, from a newly instantiated CacheManager, for all cache regions.

org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory
Uses a single JBoss Cache configuration, from an existing CacheManager bound to JNDI, for all cache regions.

A multiplexed region factory uses separate Cache instances, using optimized configurations for each cache region.

Hibernate ships with 2 multiplexed region factory implementations:

org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory
Uses separate JBoss Cache configurations, from a newly instantiated CacheManager, per cache region.

org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory
Uses separate JBoss Cache configurations, from a JNDI-bound CacheManager, see *Section 3.2.1, "The JBoss AS CacheManager Service"*, per cache region.

> **What `RegionFactory` is best to use inside JBoss AS?**
>
> Use `org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory`. It integrates cleanly with the `CacheManager` service (see *Section 3.2.1, "The JBoss AS CacheManager Service"* that is the source of JBoss Cache instances for all of the standard JBoss AS clustered services.

Depending on what class you specify as your `hibernate.cache.region.factory_class`, there are other configuration properties available that are specific to that `RegionFactory`:

7.1.1.1. Additional properties for SharedJBossCacheRegionFactory

hibernate.cache.region.jbc2.cfg.shared
The classpath or filesystem resource containing the JBoss Cache configuration settings. Default is `treecache.xml`

hibernate.cache.region.jbc2.cfg.jgroups.stacks
The classpath or filesystem resource containing the JGroups protocol stack configurations. Default is `org/hibernate/cache/jbc2/builder/jgroups-stacks.xml`

7.1.1.2. Additional properties for JndiSharedJBossCacheRegionFactory

hibernate.cache.region.jbc2.cfg.shared
JNDI name to which the shared `Cache` instance is bound. Configuring this property is required, as there is no default.

7.1.1.3. Additional properties for MultiplexedJBossCacheRegionFactory

hibernate.cache.region.jbc2.configs
The classpath or filesystem resource containing the JBoss Cache configuration settings. Default is `org/hibernate/cache/jbc2/builder/jbc2-configs.xml`.

hibernate.cache.region.jbc2.cfg.jgroups.stacks
The classpath or filesystem resource containing the JGroups protocol stack configurations. Default is `org/hibernate/cache/jbc2/builder/jgroups-stacks.xml`

hibernate.cache.region.jbc2.cfg.entity
The JBoss Cache configuration used for the entity cache region. Default is `optimistic-entity`. Alternative configurations: mvcc-entity, pessimistic-entity, mvcc-entity-repeatable,

optimistic-entity-repeatable, pessimistic-entity-repeatable. See *Section 3.2.1, "The JBoss AS CacheManager Service"*.

hibernate.cache.region.jbc2.cfg.collection
The JBoss Cache configuration used for the collection cache region. The default behavior is for the collection cache to use the same configuration as the entity cache.

hibernate.cache.region.jbc2.cfg.query
The JBoss Cache configuration used for the query cache region. The default value is `local-query`, which results in cached query results not being replicated. Alternative configurations: replicated-query

hibernate.cache.region.jbc2.cfg.ts
The JBoss Cache configuration used for the timestamp cache region. If query caching is used, the corresponding timestamp cache must be replicating, even if the query cache is non-replicating. The timestamp cache region must never share the same cache as the query cache. Default value is `timestamps-cache`.

7.1.1.4. Additional properties for JndiMultiplexedJBossCacheRegionFactory

hibernate.cache.region.jbc2.cachefactory
JNDI name to which the `CacheManager` instance is bound. Must be specified, as there is no default. Inside JBoss AS use `java:CacheManager`.

hibernate.cache.region.jbc2.cfg.entity
The JBoss Cache configuration used for the entity cache region. Default is `optimistic-entity`. Alternative configurations: mvcc-entity, pessimistic-entity, mvcc-entity-repeatable, optimistic-entity-repeatable, pessimistic-entity-repeatable. See *Section 3.2.1, "The JBoss AS CacheManager Service"*.

hibernate.cache.region.jbc2.cfg.collection
The JBoss Cache configuration used for the collection cache region. The default behavior is for the collection cache to use the same configuration as the entity cache.

hibernate.cache.region.jbc2.cfg.query
The JBoss Cache configuration used for the query cache region. The default value is `local-query`, which results in cached query results not being replicated. Alternative configurations: replicated-query

hibernate.cache.region.jbc2.cfg.ts
The JBoss Cache configuration used for the timestamp cache region. If query caching is used, the corresponding timestamp cache must be replicating, even if the query cache is non-replicating. The timestamp cache region must never share the same cache as the query cache. Default value is `timestamps-cache`.

Now, we have JBoss Cache configured to support distributed caching of EJB 3.0 entity beans. We still have to configure individual entity beans to use the cache service.

7.1.2. Configure the entity beans for caching

Next we need to configure which entities to cache. The default is to not cache anything, even with the settings shown above. We use the `@org.hibernate.annotations.Cache` annotation to tag entity beans that needs to be cached.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
public class Account implements Serializable
{
  // ... ...
}
```

A very simplified rule of thumb is that you will typically want to do caching for objects that rarely change, and which are frequently read. You can fine tune the cache for each entity bean in the appropriate JBoss Cache configuration file, e.g. jboss-cache-manager-jboss-beans.xml. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache can evict the oldest or least used objects, depending on configuration, to make room for new objects. Assuming the region_prefix specified in `persistence.xml` was `myprefix`, the default name of the cache region for the `com.mycompany.entities.Account` entity bean would be `/myprefix/com/mycompany/entities/Account`.

```
<bean name="..." class="org.jboss.cache.config.Configuration">
  ... ...
 <property name="evictionConfig">
  <bean class="org.jboss.cache.config.EvictionConfig">
   <property name="wakeupInterval">5000</property>
   <!--  Overall default -->
   <property name="defaultEvictionRegionConfig">
    <bean class="org.jboss.cache.config.EvictionRegionConfig">
     <property name="regionName">/</property>
     <property name="evictionAlgorithmConfig">
      <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
       <!-- Evict LRU node once we have more than this number of nodes -->
       <property name="maxNodes">10000</property>
       <!-- And, evict any node that hasn't been accessed in this many seconds -->
       <property name="timeToLiveSeconds">1000</property>
       <!-- Don't evict a node that's been accessed within this many seconds.
          Set this to a value greater than your max expected transaction length. -->
       <property name="minTimeToLiveSeconds">120</property>
      </bean>
     </property>
```

```
      </bean>
    </property>
    <property name="evictionRegionConfigs">
      <list>
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/myprefix/com/mycompany/entities/Account</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">10000</property>
              <property name="timeToLiveSeconds">5000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
        ... ...
      </list>
    </property>
  </bean>
 </property>
</bean>
```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached using the `defaultEvictionRegionConfig` as defined above. The @Cache annotation exposes an optional attribute “region” that lets you specify the cache region where an entity is to be stored, rather than having it be automatically be created from the fully-qualified class name of the entity class.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
public class Account implements Serializable
{
  // ... ...
}
```

The eviction configuration would then become:

```
<bean name="..." class="org.jboss.cache.config.Configuration">
   ... ...
 <property name="evictionConfig">
   <bean class="org.jboss.cache.config.EvictionConfig">
     <property name="wakeupInterval">5000</property>
     <!--  Overall default -->
```

```
    <property name="defaultEvictionRegionConfig">
      <bean class="org.jboss.cache.config.EvictionRegionConfig">
        <property name="regionName">/</property>
        <property name="evictionAlgorithmConfig">
          <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
            <property name="maxNodes">5000</property>
            <property name="timeToLiveSeconds">1000</property>
            <property name="minTimeToLiveSeconds">120</property>
          </bean>
        </property>
      </bean>
    </property>
    <property name="evictionRegionConfigs">
      <list>
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/myprefix/Account</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <property name="maxNodes">10000</property>
              <property name="timeToLiveSeconds">5000</property>
              <property name="minTimeToLiveSeconds">120</property>
            </bean>
          </property>
        </bean>
          ... ...
      </list>
    </property>
   </bean>
  </property>
</bean>
```

7.1.3. Query result caching

The EJB3 Query API also provides means for you to save the results (i.e., collections of primary keys of entity beans, or collections of scalar values) of specified queries in the second-level cache. Here we show a simple example of annotating a bean with a named query, also providing the Hibernate-specific hints that tells Hibernate to cache the query.

First, in persistence.xml you need to tell Hibernate to enable query caching:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

Next, you create a named query associated with an entity, and tell Hibernate you want to cache the results of that query:

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
@NamedQueries(
{
  @NamedQuery(
    name = "account.bybranch",
    query = "select acct from Account as acct where acct.branch = ?1",
    hints = { @QueryHint(name = "org.hibernate.cacheable", value = "true") }
  )
})
public class Account implements Serializable
{
  // ... ...
}
```

The @NamedQueries, @NamedQuery and @QueryHint annotations are all in the javax.persistence package. See the Hibernate and EJB3 documentation for more on how to use EJB3 queries and on how to instruct EJB3 to cache queries.

By default, Hibernate stores query results in JBoss Cache in a region named {region_prefix}/org/hibernate/cache/StandardQueryCache. Based on this, you can set up separate eviction handling for your query results. So, if the region prefix were set to myprefix in persistence.xml, you could, for example, create this sort of eviction handling:

```
<bean name="..." class="org.jboss.cache.config.Configuration">
    ... ...
  <property name="evictionConfig">
     <bean class="org.jboss.cache.config.EvictionConfig">
       <property name="wakeupInterval">5000</property>
       <!--  Overall default -->
       <property name="defaultEvictionRegionConfig">
         <bean class="org.jboss.cache.config.EvictionRegionConfig">
           <property name="regionName">/</property>
           <property name="evictionAlgorithmConfig">
             <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
               <property name="maxNodes">5000</property>
               <property name="timeToLiveSeconds">1000</property>
               <property name="minTimeToLiveSeconds">120</property>
             </bean>
           </property>
         </bean>
       </property>
       <property name="evictionRegionConfigs">
```

```
        <list>
          <bean class="org.jboss.cache.config.EvictionRegionConfig">
            <property name="regionName">/myprefix/Account</property>
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">10000</property>
                <property name="timeToLiveSeconds">5000</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
          <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/myprefix/org/hibernate/cache/StandardQueryCache</
property>
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">100</property>
                <property name="timeToLiveSeconds">600</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
        </list>
      </property>
    </bean>
  </property>
</bean>
```

The @NamedQuery.hints attribute shown above takes an array of vendor-specific @QueryHints as a value. Hibernate accepts the “org.hibernate.cacheRegion” query hint, where the value is the name of a cache region to use instead ofthe default /org/hibernate/cache/StandardQueryCache. For example:

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL, region = ”Account”)
@NamedQueries(
{
  @NamedQuery(
    name = "account.bybranch",
    query = "select acct from Account as acct where acct.branch = ?1",
    hints =
    {
      @QueryHint(name = "org.hibernate.cacheable", value = "true"),
```

```
    @QueryHint(name = ”org.hibernate.cacheRegion, value = ”Queries”)
   }
  )
})
public class Account implements Serializable
{
  // ... ...
}
```

The related eviction configuration:

```
<bean name="..." class="org.jboss.cache.config.Configuration">
   ... ...
  <property name="evictionConfig">
     <bean class="org.jboss.cache.config.EvictionConfig">
       <property name="wakeupInterval">5000</property>
       <!--  Overall default -->
       <property name="defaultEvictionRegionConfig">
         <bean class="org.jboss.cache.config.EvictionRegionConfig">
           <property name="regionName">/</property>
           <property name="evictionAlgorithmConfig">
             <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
               <property name="maxNodes">5000</property>
               <property name="timeToLiveSeconds">1000</property>
               <property name="minTimeToLiveSeconds">120</property>
             </bean>
           </property>
         </bean>
       </property>
       <property name="evictionRegionConfigs">
         <list>
           <bean class="org.jboss.cache.config.EvictionRegionConfig">
             <property name="regionName">/myprefix/Account</property>
             <property name="evictionAlgorithmConfig">
               <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                 <property name="maxNodes">10000</property>
                 <property name="timeToLiveSeconds">5000</property>
                 <property name="minTimeToLiveSeconds">120</property>
               </bean>
             </property>
           </bean>
           <bean class="org.jboss.cache.config.EvictionRegionConfig">
             <property name="regionName">/myprefix/Queries</property>
```

```
            <property name="evictionAlgorithmConfig">
              <bean class="org.jboss.cache.eviction.LRUAlgorithmConfig">
                <property name="maxNodes">100</property>
                <property name="timeToLiveSeconds">600</property>
                <property name="minTimeToLiveSeconds">120</property>
              </bean>
            </property>
          </bean>
          ... ...
        </list>
      </property>
    </bean>
  </property>
</bean>
```

7.2. Entity Bean in EJB 2.x

First of all, it is worth noting that clustering 2.x entity beans is a bad thing to do. Its exposes elements that generally are too fine grained for use as remote objects to clustered remote objects and introduces data synchronization problems that are non-trivial. Do NOT use EJB 2.x entity bean clustering unless you fit into the sepecial case situation of read-only, or one read-write node with read-only nodes synched with the cache invalidation services.

To use a clustered entity bean, the application does not need to do anything special, except for looking up EJB 2.x remote bean references from the clustered HA-JNDI.

To cluster EJB 2.x entity beans, you need to add the `<clustered>` element to the application's `jboss.xml` descriptor file. Below is a typical `jboss.xml` file.

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>nextgen.EnterpriseEntity</ejb-name>
      <jndi-name>nextgen.EnterpriseEntity</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
        <partition-name>DefaultPartition</partition-name>
        <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
        </home-load-balance-policy>
        <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.FirstAvailable
        </bean-load-balance-policy>
      </cluster-config>
```

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

The EJB 2.x entity beans are clustered for load balanced remote invocations. All the bean instances are synchronized to have the same contents on all nodes.

However, clustered EJB 2.x Entity Beans do not have a distributed locking mechanism or a distributed cache. They can only be synchronized by using row-level locking at the database level (see `<row-lock>` in the CMP specification) or by setting the Transaction Isolation Level of your JDBC driver to be `TRANSACTION_SERIALIZABLE`. Because there is no supported distributed locking mechanism or distributed cache Entity Beans use Commit Option "B" by default (See `standardjboss.xml` and the container configurations Clustered CMP 2.x EntityBean, Clustered CMP EntityBean, or Clustered BMP EntityBean). It is not recommended that you use Commit Option "A" unless your Entity Bean is read-only. (There are some design patterns that allow you to use Commit Option "A" with read-mostly beans. You can also take a look at the Seppuku pattern *http://dima.dhs.org/misc/readOnlyUpdates.html*. JBoss may incorporate this pattern into later versions.)

Note

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own. The MVCSoft CMP 2.0 persistence engine (see *http://www.jboss.org/jbossgroup/partners.jsp*) provides different kinds of optimistic locking strategies that can work in a JBoss cluster.

HTTP Services

HTTP session replication is used to replicate the state associated with web client sessions to other nodes in a cluster. Thus, in the event one of your nodes crashes, another node in the cluster will be able to recover. Two distinct functions must be performed:

- Session state replication
- Load-balancing of incoming invocations

State replication is directly handled by JBoss. When you run JBoss in the `all` configuration, session state replication is enabled by default. Just configure your web application as `<distributable>` in its `web.xml` (see *Section 8.2, "Configuring HTTP session state replication"*), deploy it, and its session state is automtically replicated across all JBoss instances in the cluster.

However, load-balancing is a different story; it is not handled by JBoss itself and requires an external load balancer. This function could be provided by specialized hardware switches or routers (Cisco LoadDirector for example) or by specialized software running on commodity hardware. As a very common scenario, we will demonstrate how to set up a software load balancer using Apache httpd and mod_jk.

> **Note**
>
> A load-balancer tracks HTTP requests and, depending on the session to which the request is linked, it dispatches the request to the appropriate node. This is called load-balancing with sticky-sessions or session affinity: once a session is created on a node, every future request will also be processed by that same node. Using a load-balancer that supports sticky-sessions but not configuring your web application for session replication allows you to scale very well by avoiding the cost of session state replication: each request for a session will always be handled by the same node. But in case a node dies, the state of all client sessions hosted by this node (the shopping carts, for example) will be lost and the clients will most probably need to login on another node and restart with a new session. In many situations, it is acceptable not to replicate HTTP sessions because all critical state is stored in a database or on the client. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

8.1. Configuring load balancing using Apache and mod_jk

Apache is a well-known web server which can be extended by plugging in modules. One of these modules, mod_jk, has been specifically designed to allow the forwarding of requests from Apache

to a Servlet container. Furthermore, it is also able to load-balance HTTP calls to a set of Servlet containers while maintaining sticky sessions, which is what is most interesting for us in this section.

8.1.1. Download the software

First of all, make sure that you have Apache installed. You can download Apache directly from Apache web site at `http://httpd.apache.org/`. Its installation is pretty straightforward and requires no specific configuration. As several versions of Apache exist, we advise you to use the latest stable 2.2.x version. We will consider, for the next sections, that you have installed Apache in the `APACHE_HOME` directory.

Next, download mod_jk binaries. Several versions of mod_jk exist as well. We strongly advise you to use mod_jk 1.2.x, as both mod_jk and mod_jk2 are deprecated, unsupported and no further development is going on in the community. The mod_jk 1.2.x binary can be downloaded from `http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/`. Rename the downloaded file to `mod_jk.so` and copy it under `APACHE_HOME/modules/`.

8.1.2. Configure Apache to load mod_jk

Modify APACHE_HOME/conf/httpd.conf and add a single line at the end of the file:

```
# Include mod_jk's specific configuration file
Include conf/mod-jk.conf
```

Next, create a new file named `APACHE_HOME/conf/mod-jk.conf`:

```
# Load mod_jk module
# Specify the filename of the mod_jk lib
LoadModule jk_module modules/mod_jk.so

# Where to find workers.properties
JkWorkersFile conf/workers.properties

# Where to put jk logs
JkLogFile logs/mod_jk.log

# Set the jk log level [debug/error/info]
JkLogLevel info

# Select the log format
JkLogStampFormat  "[%a %b %d %H:%M:%S %Y]"
```

```
# JkOptions indicates to send SSK KEY SIZE
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /application/* loadbalancer

# You can use external file for mount points.
# It will be checked for updates each 60 seconds.
# The format of the file is: /url=worker
# /examples/*=loadbalancer
JkMountFile conf/uriworkermap.properties

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm

# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1
</Location>
```

Please note that two settings are very important:

- The `LoadModule` directive must reference the mod_jk library you have downloaded in the previous section. You must indicate the exact same name with the "modules" file path prefix.
- The `JkMount` directive tells Apache which URLs it should forward to the mod_jk module (and, in turn, to the Servlet containers). In the above file, all requests with URL path `/application/*` are sent to the mod_jk load-balancer. This way, you can configure Apache to serve static contents (or PHP contents) directly and only use the loadbalancer for Java applications. If you only use mod_jk as a loadbalancer, you can also forward all URLs (i.e., `/*`) to mod_jk.

In addition to the `JkMount` directive, you can also use the `JkMountFile` directive to specify a mount points configuration file, which contains multiple Tomcat forwarding URL mappings. You just need to create a `uriworkermap.properties` file in the `APACHE_HOME/conf` directory. The format of the file is `/url=worker_name`. To get things started, paste the following example into the file you created:

```
# Simple worker configuration file

# Mount the Servlet context to the ajp13 worker
/jmx-console=loadbalancer
/jmx-console/*=loadbalancer
/web-console=loadbalancer
/web-console/*=loadbalancer
```

This will configure mod_jk to forward requests to `/jmx-console` and `/web-console` to Tomcat.

You will most probably not change the other settings in `mod_jk.conf`. They are used to tell mod_jk where to put its logging file, which logging level to use and so on.

8.1.3. Configure worker nodes in mod_jk

Next, you need to configure mod_jk workers file `conf/workers.properties`. This file specifies where the different Servlet containers are located and how calls should be load-balanced across them. The configuration file contains one section for each target servlet container and one global section. For a two nodes setup, the file could look like this:

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status

# Define Node1
# modify the host as your host IP or DNS name.
worker.node1.port=8009
worker.node1.host=node1.mydomain.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
# modify the host as your host IP or DNS name.
worker.node2.port=8009
worker.node2.host= node2.mydomain.com
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1,node2
```

```
worker.loadbalancer.sticky_session=1
#worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

Basically, the above file configures mod_jk to perform weighted round-robin load balancing with sticky sessions between two servlet containers (i.e. JBoss AS instances) node1 and node2 listening on port 8009.

In the `workers.properties` file, each node is defined using the `worker.XXX` naming convention where `XXX` represents an arbitrary name you choose for each of the target Servlet containers. For each worker, you must specify the host name (or IP address) and the port number of the AJP13 connector running in the Servlet container.

The `lbfactor` attribute is the load-balancing factor for this specific worker. It is used to define the priority (or weight) a node should have over other nodes. The higher this number is for a given worker relative to the other workers, the more HTTP requests the worker will receive. This setting can be used to differentiate servers with different processing power.

The `cachesize` attribute defines the size of the thread pools associated to the Servlet container (i.e. the number of concurrent requests it will forward to the Servlet container). Make sure this number does not outnumber the number of threads configured on the AJP13 connector of the Servlet container. Please review `http://jakarta.apache.org/tomcat/connectors-doc/config/workers.html` for comments on `cachesize` for Apache 1.3.x.

The last part of the `conf/workers.properties` file defines the loadbalancer worker. The only thing you must change is the `worker.loadbalancer.balanced_workers` line: it must list all workers previously defined in the same file: load-balancing will happen over these workers.

The `sticky_session` property specifies the cluster behavior for HTTP sessions. If you specify `worker.loadbalancer.sticky_session=0`, each request will be load balanced between node1 and node2; i.e., different requests for the same session will go to different servers. But when a user opens a session on one server, it is always necessary to always forward this user's requests to the same server, as long as that server is available. This is called a "sticky session", as the client is always using the same server he reached on his first request. To enable session stickiness, you need to set `worker.loadbalancer.sticky_session` to 1.

> **Note**
>
> A non-loadbalanced setup with a single node requires a `worker.list=node1` entry.

8.1.4. Configuring JBoss to work with mod_jk

Finally, we must configure the JBoss AS instances on all clustered nodes so that they can expect requests forwarded from the mod_jk loadbalancer.

On each clustered JBoss node, we have to name the node according to the name specified in `workers.properties`. For instance, on JBoss instance node1, edit the `JBOSS_HOME/server/all/deploy/jbossweb.sar/server.xml` file (replace `/all` with your own server name if necessary). Locate the `<Engine>` element and add an attribute `jvmRoute`:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
... ...
</Engine>
```

You also need to be sure the AJP connector in server.xml is enabled (i.e., uncommented). It is enabled by default.

```
<!-- An AJP 1.3 Connector on port 8009 -->
<Connector protocol="AJP/1.3" port="8009" address="${jboss.bind.address}"
   redirectPort="8443" />
```

At this point, you have a fully working Apache+mod_jk load-balancer setup that will balance call to the Servlet containers of your cluster while taking care of session stickiness (clients will always use the same Servlet container).

> **Note**
>
> For more updated information on using mod_jk 1.2 with JBoss AS, please refer to the JBoss wiki page at `http://www.jboss.org/community/wiki/UsingModjk12WithJBoss`.

8.2. Configuring HTTP session state replication

The preceding discussion has been focused on using mod_jk as a load balancer. The content of the remainder our discussion of clustering HTTP services in JBoss AS applies no matter what load balancer is used.

In *Section 8.1.3, "Configure worker nodes in mod_jk"*, we covered how to use sticky sessions to make sure that a client in a session always hits the same server node in order to maintain the session state. However, sticky sessions by themselves are not an ideal solution. If a node goes down, all its session data is lost. A better and more reliable solution is to replicate session data

across the nodes in the cluster. This way, if a server node fails or is shut down, the load balancer can fail over the next client request to any server node and obtain the same session state.

8.2.1. Enabling session replication in your application

To enable replication of your web application sessions, you must tag the application as distributable in the `web.xml` descriptor. Here's an example:

```
<?xml version="1.0"?>
<web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
        version="2.4">

   <distributable/>

</web-app>
```

You can futher configure session replication using the `replication-config` element in the `jboss-web.xml` file. However, the `replication-config` element only needs to be set if one or more of the default values described below is unacceptable. Here is an example:

```
<!DOCTYPE jboss-web PUBLIC
    -//JBoss//DTD Web Application 5.0//EN
    http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

  <replication-config>
     <cache-name>custom-session-cache</cache-name>
     <replication-trigger>SET</replication-trigger>
     <replication-granularity>ATTRIBUTE</replication-granularity>
     <replication-field-batch-mode>true</replication-field-batch-mode>
     <use-jk>false</use-jk>
     <max-unreplicated-interval>30</max-unreplicated-interval>
     <snapshot-mode>instant</snapshot-mode>
     <snapshot-interval>1000</snapshot-interval>
     <session-notification-policy>com.example.CustomPolicy</session-notification-policy>
  </replication-config>

</jboss-web>
```

All of the above configuration elements are optional and can be ommitted if the default value is acceptable. A couple are commonly used; the rest are very infrequently changed from the defaults. We'll cover the commonly used ones first.

The **`replication-trigger`** element determines when the container should consider that session data must be replicated across the cluster. The rationale for this setting is that after a mutable object stored as a session attribute is accessed from the session, in the absence of a `setAttribute` call the container has no clear way to know if the object (and hence the session state) has been modified and needs to be replicated. This element has 3 valid values:

- **SET_AND_GET** is conservative but not optimal (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. Since setting `max_unreplicated_interval` to 0 accomplishes the same thing at much lower cost, using `SET_AND_GET` makes no sense with AS 5.
- **SET_AND_NON_PRIMITIVE_GET** is conservative but will only replicate if an object of a non-primitive type has been accessed (i.e. the object is not of a well-known immutable JDK type such as `Integer`, `Long`, `String`, etc.) 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, but requires very good coding practices to ensure `setAttribute` is always called whenever a mutable object stored in the session is modified.

In all cases, calling `setAttribute` marks the session as needing replication.

The **`replication-granularity`** element determines the granularity of what gets replicated if the container determines session replication is needed. The supported values are:

- **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. However, it is not suitable for applications that store objects in different attributes that share references with each other (e.g. a `Person` object in the "husband" attribute sharing with another `Person` in the "wife" attribute a reference to an `Address` object). This is because if the attributes are separately replicated, when the session is deserialized on remote nodes the shared references will be broken.

- **FIELD** is useful if the classes stored in the session have been bytecode enhanced for use by POJO Cache. If they have been, the session management layer will detect field level changes within objects stored to the session, and will replicate only those changes. This is the most performant setting. Replication is only for individual changed data fields inside session attribute objects. Shared object references will be preserved across the cluster. Potentially most performant, but requires changes to your application (this will be discussed later).

The other elements under the `replication-config` element are much less frequently used.

The **`cacheName`** element indicates the name of the JBoss Cache configuration that should be used for storing distributable sessions and replicating them around the cluster. This element allows webapps that need different caching characteristics to specify the use of separate, differently configured, JBoss Cache instances. In AS 4 the cache to use was a server-wide configuration that could not be changed per webapp. The default value is `standard-session-cache` if the `replication-granularity` is not `FIELD`, `field-granularity-session-cache` if it is. See *Section 8.2.3, "Configuring the JBoss Cache instance used for session state replication"* for more details on JBoss Cache configuration for web tier clustering.

The **`replication-field-batch-mode`** element indicates whether you want all replication messages associated with a request to be batched into one message. Only applicable if `replication-granularity` is `FIELD`. If this is set to `true`, fine-grained changes made to objects stored in the session attribute map will replicate only when the http request is finished; otherwise they replicate as they occur. Setting this to `false` is not advised. Default is `true`.

The **`useJK`** element indicates whether the container should assume a JK-based software load balancer (e.g. mod_jk, mod_proxy, mod_cluster) is used for load balancing for this webapp. If set to `true`, the container will examine the session id associated with every request and replace the jvmRoute portion of the session id if it detects a failover.

The default value is `null` (i.e. unspecified), in which case the session manager will use the presence or absence of a jvmRoute configuration on its enclosing JBoss Web `Engine` (see *Section 8.1.4, "Configuring JBoss to work with mod_jk"*) as indicating whether JK is used.

The only real reason to set this element is to set it to `false` for a particular webapp whose URL's the JK load balancer doesn't handle. Even doing that isn't really necessary.

The **`max-unreplicated-interval`** element configures the maximum interval between requests, in seconds, after which a request will trigger replication of the session's timestamp regardless of whether the request has otherwise made the session dirty. Such replication ensures that other nodes in the cluster are aware of the most recent value for the session's timestamp and won't incorrectly expire an unreplicated session upon failover. It also results in correct values for HttpSession.getLastAccessedTime() calls following failover.

A value of `0` means the timestamp will be replicated whenever the session is accessed. A value of `-1` means the timestamp will be replicated only if some other activity during the request (e.g. modifying an attribute) has resulted in other replication work involving the session. A positive value greater than the `HttpSession.getMaxInactiveInterval()` value will be treated as a likely

misconfiguration and converted to `0`; i.e. replicate the metadata on every request. Default value is `60`.

The **`snapshot-mode`** element configures when sessions are replicated to the other nodes. Possible values are `instant` (the default) and `interval`.

The typical value, `instant`, replicates changes to the other nodes at the end of requests, using the request processing thread to perform the replication. In this case, the `snapshot-interval` property is ignored.

With `interval` mode, a background task is created that runs every `snapshot-interval` milliseconds, checking for modified sessions and replicating them.

Note that this property has no effect if `replication-granularity` is set to `FIELD`. If it is `FIELD`, `instant` mode will be used.

The **`snapshot-interval`** element defines how often (in milliseconds) the background task that replicates modified sessions should be started for this web app. Only meaningful if `snapshot-mode` is set to `interval`.

The **`session-notification-policy`** element specifies the fully qualified class name of the implementation of the `ClusteredSessionNotificationPolicy` interface that should be used to govern whether servlet specification notifications should be emitted to any registered `HttpSessionListener`, `HttpSessionAttributeListener` and/or `HttpSessionBindingListener`.

Event notifications that may make sense in a non-clustered environment may or may not make sense in a clustered environment; see *https://jira.jboss.org/jira/browse/JBAS-5778* for an example of why a notification may not be desired. Configuring an appropriate `ClusteredSessionNotificationPolicy` gives the application author fine-grained control over what notifications are issued.

In AS 5.0.0.GA the default value if not explicitly set is the `LegacyClusteredSessionNotificationPolicy`, which implements the behavior in previous JBoss versions. In the AS 5.1.0 release this was changed to `IgnoreUndeployLegacyClusteredSessionNotificationPolicy`, which implements the same behavior except for in undeployment situations, during which no `HttpSessionListener` and `HttpSessionAttributeListener` notifications are sent.

8.2.2. HttpSession Passivation and Activation

Passivation is the process of controlling memory usage by removing relatively unused sessions from memory while storing them in persistent storage. If a passivated session is requested by a client, it can be "activated" back into memory and removed from the persistent store. JBoss AS 5 supports passivation of HttpSessions from webapps whose `web.xml` includes the `distributable` tag (i.e. clustered webapps).

Passivation occurs at 3 points during the lifecycle of a web application:

- When the container requests the creation of a new session. If the number of currently active sessions exceeds a configurable limit, an attempt is made to passivate sessions to make room in memory.
- Periodically (by default every ten seconds) as the JBoss Web background task thread runs.
- When the web application is deployed and a backup copy of sessions active on other servers is acquired by the newly deploying web app's session manager.

A session will be passivated if one of the following holds true:

- The session hasn't been used in greater than a configurable maximum idle time.
- The number of active sessions exceeds a configurable maximum and the session hasn't been used in greater than a configurable minimum idle time.

In both cases, sessions are passivated on a Least Recently Used (LRU) basis.

8.2.2.1. Configuring HttpSession Passivation

Session passivation behavior is configured via the `jboss-web.xml` deployment descriptor in your webapp's `WEB-INF` directory.

```
<!DOCTYPE jboss-web PUBLIC
    -//JBoss//DTD Web Application 5.0//EN
    http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd>

<jboss-web>

  <max-active-sessions>20</max-active-sessions>
  <passivation-config>
    <use-session-passivation>true</use-session-passivation>
    <passivation-min-idle-time>60</passivation-min-idle-time>
    <passivation-max-idle-time>600</passivation-max-idle-time>
  </passivation-config>


</jboss-web>
```

- **max-active-session**

  Determines the maximum number of active sessions allowed. If the number of sessions managed by the the session manager exceeds this value and passivation is enabled, the excess will be passivated based on the configured `passivation-min-idle-time`. If after passivation

is completed (or if passivation is disabled), the number of active sessions still exceeds this limit, attempts to create new sessions will be rejected. If set to `-1` (the default), there is no limit

- **use-session-passivation**

  Determines whether session passivation will be enabled for the web application. Default is `false`.

- **passivation-min-idle-time**

  Determines the minimum time (in seconds) that a session must have been inactive before the container will consider passivating it in order to reduce the active session count below `max-active-sessions`. A value of `-1` (the default) disables passivating sessions before passivation-max-idle-time. Neither a value of `-1` nor a high value are recommended if `max-active-sessions` is set.

- **passivation-max-idle-time**

  Determines the maximum time (in seconds) that a session can be inactive before the container should attempt to passivate it to save memory. Passivation of such sessions will take place regardless of whether the active session count exceeds `max-active-sessions`. Should be less than the web.xml `session-timeout` setting. A value of `-1` (the default) disables passivation based on maximum inactivity.

Note that the number of sessions in memory includes sessions replicated from other cluster nodes that are not being accessed on this node. Be sure to account for that when setting `max-active-sessions`. Note also that the number of sessions replicated from other nodes may differ greatly depending on whether buddy replication is enabled. In an 8 node cluster where each node is handling requests from 100 users, with total replication each node will have 800 sessions in memory. With buddy replication with the default `numBuddies` setting of `1`, each node will have 200 sessions in memory.

8.2.3. Configuring the JBoss Cache instance used for session state replication

The container for a distributable web application makes use of JBoss Cache to provide HTTP session replication services around the cluster. The container integrates with the CacheManager service to obtain a reference to a JBoss Cache instance (see *Section 3.2.1, "The JBoss AS CacheManager Service"*).

The name of the JBoss Cache configuration to use is controlled by the `cacheName` element in the application's `jboss-web.xml` (see *Section 8.2.1, "Enabling session replication in your application"*). In most cases, though, this does not need to be set as the default values of `standard-session-cache` and `field-granularity-session-cache` (for applications configured for FIELD granularity) are appropriate.

The JBoss Cache configurations in the CacheManager service expose are large number of options. See *Chapter 11, JBoss Cache Configuration and Deployment* and the JBoss Cache

documentation for a more complete discussion. However, the `standard-session-cache` and `field-granularity-session-cache` configurations are already optimized for the web session replication use case, and most of the settings should not be altered. However, there are a few items that an JBoss AS administrator may wish to change:

- **cacheMode**

  By default, `REPL_ASYNC`, meaning a web request thread sending a session replication message to the cluster does not wait for responses from other cluster nodes confirming they have received and processed the message. Alternative `REPL_SYNC` offers greater guarantees that the session state was received, but at a significant performance cost. See *Section 11.1.2, "Cache Mode"*.

- **enabled** property in the **buddyReplicationConfig** section

  Set to `true` to enable buddy replication. See *Section 11.1.8, "Buddy Replication"*. Default is `false`.

- **numBuddies** property in the **buddyReplicationConfig** section

  Set to a value greater than the default `1` to increase the number of backup nodes onto which sessions are replicated. Only relevant if buddy replication is enabled. See *Section 11.1.8, "Buddy Replication"*.

- **buddyPoolName** property in the **buddyReplicationConfig** section

  A way to specify a preferred replication group when buddy replication is enabled. JBoss Cache tries to pick a buddy who shares the same pool name (falling back to other buddies if not available). Only relevant if buddy replication is enabled. See *Section 11.1.8, "Buddy Replication"*.

- **multiplexerStack**

  Name of the JGroups protocol stack the cache should use. See *Section 3.1.1, "The Channel Factory Service"*.

- **clusterName**

  Identifying name JGroups will use for this cache's channel. Only change this if you create a new cache configuration, in which case this property should have a different value from all other cache configurations.

If you wish to use a completely new JBoss Cache configuration rather than editing one of the existing ones, please see *Section 11.2.1, "Deployment Via the CacheManager Service"*.

8.3. Using FIELD level replication

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 (i.e., 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.

The first step in doing this is to identify the classes that need to be prepared. This is done via annotations. For example:

```
@org.jboss.cache.pojo.annotation.Replicable
public class Address
{
...
}
```

If you annotate a class with @Replicable, then all of its subclasses will be automatically annotated as well. Similarly, you can annotate an interface with @Replicable and all of its implementing classes will be annotated. For example:

```
@org.jboss.cache.pojo.annotation.Replicable
public class Person
{
...
}

public class Student extends Person
{
...
}
```

There is no need to annotate `Student`. POJO Cache will recognize it as @Replicable because it is a sub-class of `Person`.

JBoss AS 5 requires JDK 5 at runtime, but some users may still need to build their projects using JDK 1.4. In this case, annotating classes can be done via JDK 1.4 style annotations embedded in JavaDocs. For example:

```
/**
 * Represents a street address.
 *
 * @@org.jboss.cache.pojo.annotation.Replicable
 */
public class Address
{
...
```

```
}
```

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by POJO Cache. You need to use the JBoss AOP pre-compiler `annotationc` and post-compiler `aopc` to process the above source code before and after they are compiled by the Java compiler. The `annotationc` step is only need if the JDK 1.4 style annotations are used; if JDK 5 annotations are used it is not necessary. Here is an example of how to invoke those commands from command line.

```
$ annotationc [classpath] [source files or directories]
$ javac -cp [classpath] [source files or directories]
$ aopc [classpath] [class files or directories]
```

Please see the JBoss AOP documentation for the usage of the pre- and post-compiler. The JBoss AOP project also provides easy to use ANT tasks to help integrate those steps into your application build process.

Finally, let's see an example on how to use FIELD-level replication on those data classes. First, we see some servlet code that reads some data from the request parameters, creates a couple of objects and stores them in the session:

```
Person husband = new Person(getHusbandName(request), getHusbandAge(request));
Person wife = new Person(getWifeName(request), getWifeAge(request));
Address addr = new Address();
addr.setPostalCode(getPostalCode(request));

husband.setAddress(addr);
wife.setAddress(addr); // husband and wife share the same address!

session.setAttribute("husband", husband); // that's it.
session.setAttribute("wife", wife); // that's it.
```

Later, a different servlet could update the family's postal code:

```
Person wife = (Person)session.getAttribute("wife");
// this will update and replicate the postal code
wife.getAddress().setPostalCode(getPostalCode(request));
```

Notice that in there is no need to call `session.setAttribute()` after you make changes to the data object, and all changes to the fields are automatically replicated across the cluster.

Besides plain objects, you can also use regular Java collections of those objects as session attributes. POJO Cache automatically figures out how to handle those collections and replicate field changes in their member objects.

8.4. Using Clustered Single Sign On

JBoss supports clustered single sign-on, allowing a user to authenticate to one web application and to be recognized on all web applications that are deployed on the same virtual host, whether or not they are deployed on that same machine or on another node in the cluster. Authentication replication is handled by JBoss Cache. Clustered single sign-on support is a JBoss-specific extension of the non-clustered `org.apache.catalina.authenticator.SingleSignOn` valve that is a standard part of Tomcat and JBoss Web. Both the non-clustered and clustered versions allow users to sign on to any one of the web apps associated with a virtual host and have their identity recognized by all other web apps on the same virtual host. The clustered version brings the added benefits of enabling SSO failover and allowing a load balancer to direct requests for different webapps to different servers, while maintaining the SSO.

8.4.1. Configuration

To enable clustered single sign-on, you must add the `ClusteredSingleSignOn` valve to the appropriate `Host` elements of the `JBOSS_HOME/server/all/deploy/jbossweb.sar/server.xml` file. The valve element is already included in the standard file; you just need to uncomment it. The valve configuration is shown here:

```
<Valve className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn" />
```

The element supports the following attributes:

- **className** is a required attribute to set the Java class name of the valve implementation to use. This must be set to `org.jboss.web.tomcat.service.sso.ClusteredSingleSign`.
- **cacheConfig** is the name of the cache configuration (see *Section 3.2.1, "The JBoss AS CacheManager Service"*) to use for the clustered SSO cache. Default is `clustered-sso`.
- **treeCacheName** is deprecated; use `cacheConfig`. Specifies a JMX ObjectName of the JBoss Cache MBean to use for the clustered SSO cache. If no cache can be located from the CacheManager service using the value of `cacheConfig`, an attempt to locate an mbean registered in JMX under this ObjectName will be made. Default value is `jboss.cache:service=TomcatClusteringCache`.
- **cookieDomain** is used to set the host domain to be used for sso cookies. See *Section 8.4.4, "Configuring the Cookie Domain"* for more. Default is `"/"`.
- **maxEmptyLife** is the maximum number of seconds an SSO with no active sessions will be usable by a request. The clustered SSO valve tracks what cluster nodes are managing sessions

related to an SSO. A positive value for this attribute allows proper handling of shutdown of a node that is the only one that had handled any of the sessions associated with an SSO. The shutdown invalidates the local copy of the sessions, eliminating all sessions from the SSO. If maxEmptyLife were zero, the SSO would terminate along with the local session copies. But, backup copies of the sessions (if they are from clustered webapps) are available on other cluster nodes. Allowing the SSO to live beyond the life of its managed sessions gives the user time to make another request which can fail over to a different cluster node, where it activates the the backup copy of the session. Default is `1800`, i.e. 30 minutes.

- **processExpiresInterval** is the minimum number of seconds between efforts by the valve to find and invalidate SSO's that have exceeded their 'maxEmptyLife'. Does not imply effort will be spent on such cleanup every 'processExpiresInterval', just that it won't occur more frequently than that. Default is `60`.
- **requireReauthentication** is a flag to determine whether each request needs to be reauthenticated to the security *Realm*. If "true", this Valve uses cached security credentials (username and password) to reauthenticate to the JBoss Web security *Realm* each request associated with an SSO session. If `false`, the valve can itself authenticate requests based on the presence of a valid SSO cookie, without rechecking with the *Realm*. Setting to `true` can allow web applications with different `security-domain` configurations to share an SSO. Default is `false`.

8.4.2. SSO Behavior

The user will not be challenged as long as he accesses only unprotected resources in any of the web applications on the virtual host.

Upon access to a protected resource in any web app, the user will be challenged to authenticate, using the login method defined for the web app.

Once authenticated, the roles associated with this user will be utilized for access control decisions across all of the associated web applications, without challenging the user to authenticate themselves to each application individually.

If the web application invalidates a session (by invoking the `javax.servlet.http.HttpSession.invalidate()` method), the user's sessions in all web applications will be invalidated.

A session timeout does not invalidate the SSO if other sessions are still valid.

8.4.3. Limitations

There are a number of known limitations to this Tomcat valve-based SSO implementation:

- Only useful within a cluster of JBoss servers; SSO does not propagate to other resources.
- Requires use of container managed authentication (via `<login-config>` element in `web.xml`)

- Requires cookies. SSO is maintained via a cookie and URL rewriting is not supported.
- Unless `requireReauthentication` is set to `true`, all web applications configured for the same SSO valve must share the same JBoss Web `Realm` and JBoss Security `security-domain`. This means:
  - In `server.xml` you can nest the `Realm` element inside the `Host` element (or the surrounding `Engine` element), but not inside a `context.xml` packaged with one of the involved web applications.
  - The `security-domain` configured in `jboss-web.xml` or `jboss-app.xml` must be consistent for all of the web applications.
  - Even if you set `requireReauthentication` to `true` and use a different `security-domain` (or, less likely, a different `Realm`) for different webapps, the varying security integrations must all accept the same credentials (e.g. username and password).

8.4.4. Configuring the Cookie Domain

As noted above the SSO valve supports a `cookieDomain` configuration attribute. This attribute allows configuration of the SSO cookie's domain (i.e. the set of hosts to which the browser will present the cookie). By default the domain is `"/"`, meaning the browser will only present the cookie to the host that issued it. The `cookieDomain` attribute allows the cookie to be scoped to a wider domain.

For example, suppose we have a case where two apps, with URLs `http://app1.xyz.com` and `http://app2.xyz.com`, that wish to share an SSO context. These apps could be running on different servers in a cluster or the virtual host with which they are associated could have multiple aliases. This can be supported with the following configuration:

```
<Valve className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn"
       cookieDomain="xyz.com" />
```

JBoss Messaging Clustering Notes

9.1. Unique server peer id

JBoss Messaging clustering should work out of the box in the *all* configuration with no configuration changes. It is however crucial that every node is assigned a unique server id.

Every node deployed must have a unique id, including those in a particular LAN cluster, and also those only linked by message bridges.

9.2. Clustered destinations

JBoss Messaging clusters JMS queues and topics transparently across the cluster. Messages sent to a distributed queue or topic on one node are consumable on other nodes. To designate that a particular destination is clustered simply set the clustered attribute in the destination deployment descriptor to true.

JBoss Messaging balances messages between nodes, catering for faster or slower consumers to efficiently balance processing load across the cluster.

If you do not want message redistribution between nodes, but still want to retain the other characteristics of clustered destinations, you can specify the attribute `ClusterPullConnectionFactoryName` on the server peer.

9.3. Clustered durable subs

JBoss Messaging durable subscriptions can also be clustered. This means multiple subscribers can consume from the same durable subscription from different nodes of the cluster. A durable subscription will be clustered if it's topic is clustered.

9.4. Clustered temporary destinations

JBoss Messaging also supports clustered temporary topics and queues. All temporary topics and queues will be clustered if the post office is clustered.

9.5. Non clustered servers

If you don't want your nodes to participate in a cluster, or only have one non clustered server you can set the clustered attribute on the postoffice to `false`.

9.6. Message ordering in the cluster

If you wish to apply strict JMS ordering to messages, such that a particular JMS consumer consumes messages in the same order as they were produced by a particular producer, you can set the `DefaultPreserveOrdering` attribute in the server peer to `true`. By default this is false.

> **Note**
>
> The side effect of setting this to true is that messages cannot be distributed as freely around the cluster.

9.7. Idempotent operations

If the call to send a persistent message to a persistent destination returns successfully with no exception, then you can be sure that the message was persisted. However if the call doesn't return successfully e.g. if an exception is thrown, then you *can't be sure the message wasn't persisted*. This is because the failure might have occurred after persisting the message but before writing the response to the caller. This is a common attribute of any RPC type call: You can't tell by the call not returning that the call didn't actually succeed. Whether it's a web services call, a HTTP get request, an EJB invocation the same applies. The trick is to code your application so your operations are *idempotent* i.e. they can be repeated without getting the system into an inconsistent state. With a message system you can do this on the application level, by checking for duplicate messages, and discarding them if they arrive. Duplicate checking is a very powerful technique that can remove the need for XA transactions in many cases.

9.7.1. Clustered connection factories

If the supportsLoadBalancing attribute of the connection factory is set to true then consecutive create connection attempts will round robin between available servers. The first node to try is chosen randomly.

If the supportsFailover attribute of the connection factory is set to true then automatic failover is enabled. This will automatically failover from one server to another, transparently to the user, in case of failure.

If automatic failover is not required or you wish to do manual failover (JBoss MQ style) this can be set to false, and you can supply a standard JMS ExceptionListener on the connection which will be called in case of connection failure. You would then need to manually close the connection, lookup a new connection factory from HA JNDI and recreate the connection.

Part III. JGroups and JBoss Cache

JGroups Services

JGroups provides the underlying group communication support for JBoss AS clusters. The way the AS's clustered services interact with JGroups was covered previously in *Section 3.1, "Group Communication with JGroups"*. The focus of this chapter is on the details, particularly configuration details and troubleshooting tips. This chapter is not intended to be a complete set of JGroups documentation; we recommend that users interested in in-depth JGroups knowledge also consult:

- The JGroups project documentation at *http://jgroups.org/ug.html*
- The JGroups wiki pages at jboss.org, rooted at *https://www.jboss.org/community/wiki/JGroups*

The first section of this chapter covers the many JGroups configuration options in considerable detail. Readers should understand that JBoss AS ships with a reasonable set of default JGroups configurations. Most applications just work out of the box with the default configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements.

10.1. Configuring a JGroups Channel's Protocol Stack

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. Communication occurs over a communication channel. The channel built up from a stack of network communication "protocols", each of which is responsible for adding a particular capability to the overall behavior of the channel. Key capabilities provided by various protocols include, among others, transport, cluster discovery, message ordering, loss-less message delivery, detection of failed peers, and cluster membership management services.

Figure 10.1, "Protocol stack in JGroups" shows a conceptual cluster with each member's channel composed of a stack of JGroups protocols.

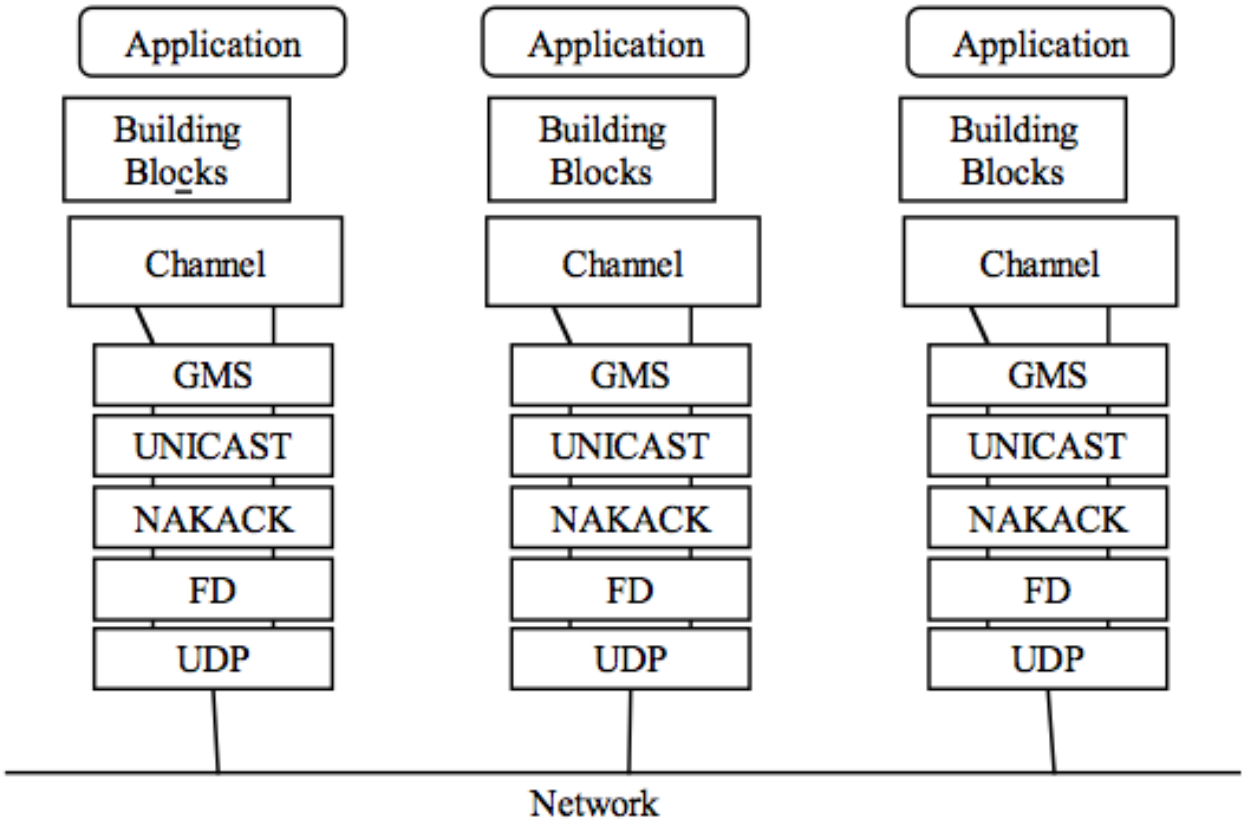

Figure 10.1. Protocol stack in JGroups

In this section of the chapter, we look into some of the most commonly used protocols, with the protocols organized by the type of behavior they add to the overall channel. For each protocol, we discuss a few key configuration attributes exposed by the protocol, but generally speaking changing configuration attributes is a matter for experts only. More important for most readers will be to get a general understanding of the purpose of the various protocols.

The JGroups configurations used in the AS appear as nested elements in the `$JBOSS_HOME/server/all/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file. This file is parsed by the `ChannelFactory` service, which uses the contents to provide appropriately configured channels to the AS clustered services that need them. See *Section 3.1.1, "The Channel Factory Service"* for more on the `ChannelFactory` service.

Following is an example protocol stack configuration from `jgroups-channelfactory-stacks.xml`:

```
<stack name="udp-async"
        description="UDP-based stack, optimized for high performance for
                asynchronous RPCs (enable_bundling=true)">
    <config>
      <UDP
```

```
    singleton_name="udp-async"
    mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
    mcast_addr="${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="true"
    max_bundle_size="64000"
    max_bundle_timeout="30"
    ip_ttl="${jgroups.udp.ip_ttl:2}"
    thread_naming_pattern="cl"
    timer.num_threads="12"
    enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"
    diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
    diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

    thread_pool.enabled="true"
    thread_pool.min_threads="8"
    thread_pool.max_threads="200"
    thread_pool.keep_alive_time="5000"
    thread_pool.queue_enabled="true"
    thread_pool.queue_max_size="1000"
    thread_pool.rejection_policy="discard"

    oob_thread_pool.enabled="true"
    oob_thread_pool.min_threads="8"
    oob_thread_pool.max_threads="200"
    oob_thread_pool.keep_alive_time="1000"
    oob_thread_pool.queue_enabled="false"
    oob_thread_pool.rejection_policy="discard"/>
  <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"/>
  <BARRIER/>
  <pbcast.NAKACK use_mcast_xmit="true" 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"/>
        <VIEW_SYNC avg_send_interval="10000"/>
        <pbcast.GMS print_local_addr="true" join_timeout="3000"
                shun="true"
                view_bundling="true"
                view_ack_collection_timeout="5000"
                resume_task_timeout="7500"/>
        <FC max_credits="2000000" min_threshold="0.10"
           ignore_synchronous_response="true"/>
        <FRAG2 frag_size="60000"/>
        <!-- pbcast.STREAMING_STATE_TRANSFER/ -->
        <pbcast.STATE_TRANSFER/>
        <pbcast.FLUSH timeout="0" start_flush_timeout="10000"/>
      </config>
    </stack>
```

All the JGroups configuration data is contained in the <config> element. This information is used to configure a JGroups Channel; the Channel is conceptually similar to a socket, and manages communication between peers in a cluster. Each element inside the <config> element defines a particular JGroups Protocol; each Protocol performs one function, and the combination of those functions is what defines the characteristics of the overall Channel. In the next several sections, we will dig into the commonly used protocols and their options and explain exactly what they mean.

10.1.1. Common Configuration Properties

The following configuration property is exposed by all of the JGroups protocols discussed below:

- `stats` whether the protocol should gather runtime statistics on its operations that can be exposed via tools like the AS's administration console or the JGroups Probe utility. What, if any, statistics are gathered depends on the protocol. Default is `true`.

> **Note**
>
> All of the protocols in the versions of JGroups used in JBoss AS 3.x and 4.x exposed `down_thread` and `up_thread` attributes. The JGroups version used in AS 5 and later no longer uses those attributes, and a WARN message will be written to the server log if they are configured for any protocol.

10.1.2. Transport Protocols

The transport protocols are responsible for actually sending messages on the network and receiving them from the network. They also manage the pools of threads that are used to

deliver incoming messages up the protocol stack. JGroups supports UDP, TCP, and TUNNEL as transport protocols.

> **Note**
>
> The `UDP`, `TCP`, and `TUNNEL` protocols are mutually exclusive. You can only have one transport protocol in each JGroups `config` element

10.1.2.1. UDP configuration

UDP is the preferred protocol for JGroups. UDP uses multicast (or, in an unusual configuration, multiple unicasts) to send and receive messages. If you choose UDP as the transport protocol for your cluster service, you need to configure it in the `UDP` sub-element in the JGroups `config` element. Here is an example.

```
<UDP
  singleton_name="udp-async"
  mcast_port="${jboss.jgroups.udp_async.mcast_port:45689}"
  mcast_addr="${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="true"
  max_bundle_size="64000"
  max_bundle_timeout="30"
  ip_ttl="${jgroups.udp.ip_ttl:2}"
  thread_naming_pattern="cl"
  timer.num_threads="12"
  enable_diagnostics="${jboss.jgroups.enable_diagnostics:true}"
  diagnostics_addr="${jboss.jgroups.diagnostics_addr:224.0.0.75}"
  diagnostics_port="${jboss.jgroups.diagnostics_port:7500}"

  thread_pool.enabled="true"
  thread_pool.min_threads="8"
  thread_pool.max_threads="200"
  thread_pool.keep_alive_time="5000"
  thread_pool.queue_enabled="true"
  thread_pool.queue_max_size="1000"
  thread_pool.rejection_policy="discard"
```

```
        oob_thread_pool.enabled="true"
        oob_thread_pool.min_threads="8"
        oob_thread_pool.max_threads="200"
        oob_thread_pool.keep_alive_time="1000"
        oob_thread_pool.queue_enabled="false"
        oob_thread_pool.rejection_policy="discard"/>
```

The available attributes in the above JGroups configuration are listed discussed below. First, we discuss the attributes that are particular to the UDP transport protocol. Then we will cover those attributes shown above that are also used by the TCP and TUNNEL transport protocols.

The attributes particular to UDP are:

- **ip_mcast** specifies whether or not to use IP multicasting. The default is `true`. If set to false, for messages to the entire group UDP will send n unicast packets rather than 1 multicast packet. Either way, packets are UDP datagrams.
- **mcast_addr** specifies the multicast address (class D) for communicating with the group (i.e., the cluster). The standard protocol stack configurations in JBoss AS use the value of system property `jboss.partition.udpGroup`, if set, as the value for this attribute. Using the `-u` command line switch when starting JBoss AS sets that value. See *Section 10.2.2, "Isolating JGroups Channels"* for how to use this configuration attribute to ensure JGroups channels are properly isolated from one another. If this attribute is omitted, the default is `228.8.8.8`.
- **mcast_port** specifies the port to use for multicast communication with the group. See *Section 10.2.2, "Isolating JGroups Channels"* for how to use this configuration attribute to ensure JGroups channels are properly isolated from one another. If this attribute is omitted, the default is `45566`.
- **mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_buf_size** define socket receive and send buffer sizes that JGroups will request from the operating system. It is good to have a large receive buffer size, so packets are less likely to get dropped due to buffer overflow. Note, however, that the size of socket buffers is limited by OS limits, so actually obtaining the desired buffer may require OS-level configuration. See *Section 10.2.3, "Improving UDP Performance by Configuring OS UDP Buffer Limits"* for further details.
- **bind_port** specifies the port to which the unicast receive socket should be bound. The default is `0`; i.e. use an ephemeral port.
- **port_range** specifies the number of ports to try if the port identified by `bind_port` is not available. The default is `1`, meaning only try to bind to `bind_port`.
- **ip_ttl** specifies time-to-live for IP Multicast packets. TTL is the commonly used term in multicast networking, but is actually something of a misnomer, since the value here refers to how many network hops a packet will be allowed to travel before networking equipment will drop it.

- **tos** specifies the traffic class for sending unicast and multicast datagrams.

The attributes that are common to all transport protocols, and thus have the same meanings when used with TCP or TUNNEL, are:

- **singleton_name** provides a unique name for this transport protocol configuration. Used by the AS `ChannelFactory` to support sharing of a transport protocol instance by different channels that use the same transport protocol configuration. See *Section 3.1.2, "The JGroups Shared Transport"*.

- **bind_addr** specifies the interface on which to receive and send messages. By default JGroups uses the value of system property `jgroups.bind_addr`, which in turn can be easily set via the `-b` command line switch. See *Section 10.2, "Key JGroups Configuration Tips"* for more on binding JGroups sockets.

- **receive_on_all_interfaces** specifies whether this node should listen on all interfaces for multicasts. The default is `false`. It overrides the `bind_addr` property for receiving multicasts. However, `bind_addr` (if set) is still used to send multicasts.

- **send_on_all_interfaces** specifies whether this node send UDP packets via all the NICs if you have a multi NIC machine. This means that the same multicast message is sent N times, so use with care.

- **receive_interfaces** specifies a list of of interfaces on which to receive multicasts. The multicast receive socket will listen on all of these interfaces. This is a comma-separated list of IP addresses or interface names. E.g. "`192.168.5.1,eth1,127.0.0.1`".

- **send_interfaces** specifies a list of of interfaces via which to send multicasts. The multicast s ender socket will send on all of these interfaces. This is a comma-separated list of IP addresses or interface names. E.g. "`192.168.5.1,eth1,127.0.0.1`".This means that the same multicast message is sent N times, so use with care.

- **enable_bundling** specifies whether to enable message bundling. If `true`, the tranpsort protocol would queue outgoing messages until `max_bundle_size` bytes have accumulated, or `max_bundle_time` milliseconds have elapsed, whichever occurs first. Then the transport protocol would bundle queued messages into one large message and send it. The messages are unbundled at the receiver. The default is `false`. Message bundling can have significant performance benefits for channels that are used for high volume sending of messages where the sender does not block waiting for a response from recipients (e.g. a JBoss Cache instance configured for REPL_ASYNC.) It can add considerable latency to applications where senders need to block waiting for responses, so it is not recommended for usages like JBoss Cache REPL_SYNC.

- **loopback** specifies whether the thread sending a message to the group should itself carry the message back up the stack for delivery. (Messages sent to the group are always delivered to the sending node as well.) If `false` the sending thread does not carry the message; rather the

transport protocol waits to read the message off the network and uses one of the message delivery pool threads to deliver it. The default is `false`, however the current recommendation is to always set this to `true` in order to ensure the channel receives its own messages in case the network interface goes down.

- **discard_incompatible_packets** specifies whether to discard packets sent by peers using a different JGroups version. Each message in the cluster is tagged with a JGroups version. When a message from a different version of JGroups is received, it will be silently discarded if this is set to true, otherwise a warning will be logged. In no case will the message be delivered. The default is `false`
- **enable_diagnostics** specifies that the transport should open a multicast socket on address `diagnostics_addr` and port `diagnostics_port` to listen for diagnostic requests sent by JGroups' ***Probe** utility* [http://www.jboss.org/community/wiki/Probe].
- The various **thread_pool** attributes configure the behavior of the pool of threads JGroups uses to carry ordinary incoming messages up the stack. The various attributes end up providing the constructor arguments for an instance of `java.util.concurrent.ThreadPoolExecutorService`. In the example above, the pool will have a core (i.e. minimum) size of 8 threads, and a maximum size of 200 threads. If more than 8 pool threads have been created, a thread returning from carrying a message will wait for up to 5000 ms to be assigned a new message to carry, after which it will terminate. If no threads are available to carry a message, the (separate) thread reading messages off the socket will place messages in a queue; the queue will hold up to 1000 messages. If the queue is full, the thread reading messages off the socket will discard the message.
- The various **oob_thread_pool** attributes are similar to the **thread_pool** attributes in that they configure a `java.util.concurrent.ThreadPoolExecutorService` used to carry incoming messages up the protocol stack. In this case, the pool is used to carry a special type of message known as an "Out-Of-Band" message, OOB for short. OOB messages are exempt from the ordered-delivery requirements of protocols like NAKACK and UNICAST and thus can be delivered up the stack even if NAKACK or UNICAST are queueing up messages from a particular sender. OOB messages are often used internally by JGroups protocols and can be used applications as well. JBoss Cache in REPL_SYNC mode, for example, uses OOB messages for the second phase of its two-phase-commit protocol.

10.1.2.2. TCP configuration

Alternatively, a JGroups-based cluster can also work over TCP connections. Compared with UDP, TCP generates more network traffic when the cluster size increases. TCP is fundamentally a unicast protocol. To send multicast messages, JGroups uses multiple TCP unicasts. To use TCP as a transport protocol, you should define a `TCP` element in the JGroups `config` element. Here is an example of the `TCP` element.

```
<TCP singleton_name="tcp"
```

```
    start_port="7800" end_port="7800"/>
```

Below are the attributes that are specific to the `TCP` protocol.

- **start_port, end_port** define the range of TCP ports the server should bind to. The server socket is bound to the first available port beginning with `start_port`. If no available port is found (e.g., because of other sockets already using the ports) before the `end_port`, the server throws an exception. If no `end_port` is provided or `end_port < start_port` then there is no upper limit on the port range. If `start_port == end_port`, then we force JGroups to use the given port (start fails if port is not available). The default is 7800. If set to 0, then the operating system will pick a port. Please, bear in mind that setting it to 0 will work only if we use MPING or TCPGOSSIP as discovery protocol because `TCCPING` requires listing the nodes and their corresponding ports.
- **bind_port** in TCP is just an alias for `start_port`; if configured internally it sets `start_port`.
- **recv_buf_size, send_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **conn_expire_time** specifies the time (in milliseconds) after which a connection can be closed by the reaper if no traffic has been received.
- **reaper_interval** specifies interval (in milliseconds) to run the reaper. If both values are 0, no reaping will be done. If either value is > 0, reaping will be enabled. By default, reaper_interval is 0, which means no reaper.
- **sock_conn_timeout** specifies max time in millis for a socket creation. When doing the initial discovery, and a peer hangs, don't wait forever but go on after the timeout to ping other members. Reduces chances of *not* finding any members at all. The default is 2000.
- **use_send_queues** specifies whether to use separate send queues for each connection. This prevents blocking on write if the peer hangs. The default is true.
- **external_addr** specifies external IP address to broadcast to other group members (if different to local address). This is useful when you have use (Network Address Translation) NAT, e.g. a node on a private network, behind a firewall, but you can only route to it via an externally visible address, which is different from the local address it is bound to. Therefore, the node can be configured to broadcast its external address, while still able to bind to the local one. This avoids having to use the TUNNEL protocol, (and hence a requirement for a central gossip router) because nodes outside the firewall can still route to the node inside the firewall, but only on its external address. Without setting the external_addr, the node behind the firewall will broadcast its private address to the other nodes which will not be able to route to it.
- **skip_suspected_members** specifies whether unicast messages should not be sent to suspected members. The default is true.

- **tcp_nodelay** specifies TCP_NODELAY. TCP by default nagles messages, that is, conceptually, smaller messages are bundled into larger ones. If we want to invoke synchronous cluster method calls, then we need to disable nagling in addition to disabling message bundling (by setting `enable_bundling` to false). Nagling is disabled by setting `tcp_nodelay` to true. The default is false.

> **Note**
>
> All of the attributes common to all protocols discussed in the UDP protocol section also apply to TCP.

10.1.2.3. TUNNEL configuration

The TUNNEL protocol uses an external router process to send messages. The external router is a Java process running the `org.jgroups.stack.GossipRouter` main class. Each node has to register with the router. All messages are sent to the router and forwarded on to their destinations. The TUNNEL approach can be used to setup communication with nodes behind firewalls. A node can establish a TCP connection to the GossipRouter through the firewall (you can use port 80). The same connection is used by the router to send messages to nodes behind the firewall as most firewalls do not permit outside hosts to initiate a TCP connection to a host inside the firewall. The TUNNEL configuration is defined in the TUNNEL element in the JGroups config element. Here is an example..

```
<TUNNEL  singleton_name="tunnel"
         router_port="12001"
         router_host="192.168.5.1"/>
```

The available attributes in the `TUNNEL` element are listed below.

- **router_host** specifies the host on which the GossipRouter is running.
- **router_port** specifies the port on which the GossipRouter is listening.
- **reconnect_interval** specifies the interval in ms at which TUNNEL will attempt to connect to the GossipRouter if the connection is not established. Default is `5000`.

> **Note**
>
> All of the attributes common to all protocols discussed in the UDP protocol section also apply to TUNNEL.

10.1.3. Discovery Protocols

When a channel on one node connects it needs to discover what other nodes have compatible channels running and which of those nodes is currently serving as the "coordinator" responsible for allowing new nodes to join the group. Discovery protocols are used to discover active nodes in the cluster and determine which is the coordinator. This information is then provided to the group membership protocol (GMS, see *Section 10.1.6, "Group Membership (GMS)"*) which communicates with the coordinator node's GMS to bring the newly connecting node into the group.

Discovery protocols also help merge protocols (see *Section 10.1.11, "Merging (MERGE2)"* to detect cluster-split situations.

Since the discovery protocols sit on top of the transport protocol, you can choose to use different discovery protocols based on your transport protocol. These are also configured as sub-elements in the JGroups `config` element.

10.1.3.1. PING

PING is a discovery protocol that works by either multicasting PING requests to an IP multicast address or connecting to a gossip router. As such, PING normally sits on top of the UDP or TUNNEL transport protocols. Each node responds with a packet {C, A}, where C=coordinator's address and A=own address. After timeout milliseconds or num_initial_members replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by). If nobody responds, we assume we are the first member of a group.

Here is an example PING configuration for IP multicast.

```
<PING timeout="2000"
   num_initial_members="3"/>
```

Here is another example PING configuration for contacting a Gossip Router.

```
<PING gossip_host="localhost"
     gossip_port="1234"
     timeout="2000"
     num_initial_members="3"/>
```

The available attributes in the `PING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.

- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **gossip_host** specifies the host on which the GossipRouter is running.
- **gossip_port** specifies the port on which the GossipRouter is listening on.
- **gossip_refresh** specifies the interval (in milliseconds) for the lease from the GossipRouter. The default is 20000.
- **initial_hosts** is a comma-separated list of addresses/ports (e.g., `host1[12345],host2[23456]`) which are pinged for discovery. Default is `null`, meaning multicast discovery should be used. If `initial_hosts` is specified, *all* possible cluster members must be listed, not just a few "well known hosts"; otherwise discovery of cluster splits by MERGE2 will not work reliably.

If both `gossip_host` and `gossip_port` are defined, the cluster uses the GossipRouter for the initial discovery. If the `initial_hosts` is specified, the cluster pings that static list of addresses for discovery. Otherwise, the cluster uses IP multicasting for discovery.

> **Note**
>
> The discovery phase returns when the `timeout` ms have elapsed or the `num_initial_members` responses have been received.

10.1.3.2. TCPGOSSIP

The TCPGOSSIP protocol only works with a GossipRouter. It works essentially the same way as the PING protocol configuration with valid `gossip_host` and `gossip_port` attributes. It works on top of both UDP and TCP transport protocols. Here is an example.

```
<TCPGOSSIP timeout="2000"
      num_initial_members="3"
      initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"/>
```

The available attributes in the `TCPGOSSIP` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-separated list of addresses/ports (e.g., `host1[12345],host2[23456]`) of GossipRouters to register with.

10.1.3.3. TCPPING

The TCPPING protocol takes a set of known members and pings them for discovery. This is essentially a static configuration. It works on top of TCP. Here is an example of the `TCPPING` configuration element in the JGroups `config` element.

```
<TCPPING timeout="2000"
    num_initial_members="3"/
    initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
    port_range="3">
```

The available attributes in the `TCPPING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2.
- **initial_hosts** is a comma-separated list of addresses/ports (e.g., `host1[12345],host2[23456]`) which are pinged for discovery. *All* possible cluster members must be listed, not just a few "well known hosts"; otherwise discovery of cluster splits by MERGE2 will not work reliably.
- **port_range** specifies the number of consecutive ports to be probed when getting the initial membership, starting with the port specified in the initial_hosts parameter. Given the current values of port_range and initial_hosts above, the TCPPING layer will try to connect to hosta[2300], hosta[2301], hosta[2302], hostb[3400], hostb[3401], hostb[3402], hostc[4500], hostc[4501], hostc[4502]. This configuration option allows for multiple possible ports on the same host to be pinged without having to spell out all of the combinations. If in your TCP protocol configuration your `end_port` is greater than your `start_port`, using a TCPPING `port_range` equal to the difference is advised in order to ensure a node is pinged no matter which port in the allowed range it ended up bound to.

10.1.3.4. MPING

MPING uses IP multicast to discover the initial membership. Unlike the other discovery protocols, which delegate the sending and receiving of discovery messages on the network to the transport protocol, MPING handles opens its own sockets to send and receive multicast discovery messages. As a result it can be used with all transports. But, it usually is used in combination with TCP. TCP usually requires TCPPING, which has to list all possible group members explicitly, while MPING doesn't have this requirement. The typical use case for MPING is when we want TCP for regular message transport, but UDP multicasting is allowed for discovery.

```
```

```
<MPING timeout="2000"
   num_initial_members="3"
   bind_to_all_interfaces="true"
   mcast_addr="228.8.8.8"
   mcast_port="7500"
   ip_ttl="8"/>
```

The available attributes in the `MPING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses. The default is 3000.
- **num_initial_members** specifies the maximum number of responses to wait for unless timeout has expired. The default is 2..
- **bind_addr** specifies the interface on which to send and receive multicast packets. By default JGroups uses the value of system property `jgroups.bind_addr`, which in turn can be easily set via the `-b` command line switch. See *Section 10.2, "Key JGroups Configuration Tips"* for more on binding JGroups sockets.
- **bind_to_all_interfaces** overrides the `bind_addr` and uses all interfaces in multihome nodes.
- **mcast_addr, mcast_port, ip_ttl** attributes are the same as related attributes in the UDP protocol configuration.

10.1.4. Failure Detection Protocols

The failure detection protocols are used to detect failed nodes. Once a failed node is detected, a suspect verification phase can occur after which, if the node is still considered dead, the cluster updates its membership view so that further messages are not sent to the failed node and the service using JGroups is aware the node is no longer part of the cluster. The failure detection protocols are configured as sub-elements in the JGroups `config` element.

10.1.4.1. FD

FD is a failure detection protocol based on heartbeat messages. This protocol requires each node to periodically send an are-you-alive message to its neighbor. If the neighbor fails to respond, the calling node sends a SUSPECT message to the cluster. The current group coordinator can optionally double check whether the suspected node is indeed dead (see VERIFY_SUSPECT below) after which, if the node is still considered dead, it updates the cluster's membership view. Here is an example FD configuration.

```
<FD timeout="6000"
   max_tries="5"
```

```
shun="true"/>
```

The available attributes in the `FD` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for the responses to the are-you-alive messages. The default is 3000.
- **max_tries** specifies the number of missed are-you-alive messages from a node before the node is suspected. The default is 2.
- **shun** specifies whether a failed node will be shunned, i.e. not allowed to send messages to the group without formally rejoining. A shunned node would have to re-join the cluster through the discovery process. JGroups allows applications to configure a channel such that shunning leads to automatic rejoins and state transfer. This the default behavior within JBoss Application Server.

> **Note**
>
> Regular traffic from a node counts as if it is a heartbeat response. So, the are-you-alive messages are only sent when there is no regular traffic to the node for some time.

10.1.4.2. FD_SOCK

FD_SOCK is a failure detection protocol based on a ring of TCP sockets created between group members. Each member in a group connects to its neighbor (last member connects to first) thus forming a ring. Member B is suspected when its neighbor A detects an abnormally closed TCP socket (presumably due to a node B crash). However, if a member B is about to leave gracefully, it lets its neighbor A know, so that it does not become suspected. The simplest FD_SOCK configuration does not take any attribute. You can just declare an empty `FD_SOCK` element in JGroups's `config` element.

```
<FD_SOCK/>
```

There available attributes in the `FD_SOCK` element are listed below.

- **bind_addr** specifies the interface to which the server socket should be bound. By default JGroups uses the value of system property `jgroups.bind_addr`, which in turn can be easily set via the `-b` command line switch. See *Section 10.2, "Key JGroups Configuration Tips"* for more on binding JGroups sockets.

10.1.4.3. VERIFY_SUSPECT

This protocol verifies whether a suspected member is really dead by pinging that member once again. This verification is performed by the coordinator of the cluster. The suspected member is dropped from the cluster group if confirmed to be dead. The aim of this protocol is to minimize false suspicions. Here's an example.

```
<VERIFY_SUSPECT timeout="1500"/>
```

The available attributes in the VERIFY_SUSPECT element are listed below.

- **timeout** specifies how long to wait for a response from the suspected member before considering it dead.

10.1.4.4. FD versus FD_SOCK

FD and FD_SOCK, each taken individually, do not provide a solid failure detection layer. Let's look at the the differences between these failure detection protocols to understand how they complement each other:

- *FD*
  - An overloaded machine might be slow in sending are-you-alive responses.
  - A member will be suspected when suspended in a debugger/profiler.
  - Low timeouts lead to higher probability of false suspicions and higher network traffic.
  - High timeouts will not detect and remove crashed members for some time.
- *FD_SOCK*:
  - Suspended in a debugger is no problem because the TCP connection is still open.
  - High load no problem either for the same reason.
  - Members will only be suspected when TCP connection breaks
  - So hung members will not be detected.
  - Also, a crashed switch will not be detected until the connection runs into the TCP timeout (between 2-20 minutes, depending on TCP/IP stack implementation).

The aim of a failure detection layer is to report promptly real failures and yet avoid false suspicions. There are two solutions:

1. By default, JGroups configures the FD_SOCK socket with KEEP_ALIVE, which means that TCP sends a heartbeat on socket on which no traffic has been received in 2 hours. If a host

crashed (or an intermediate switch or router crashed) without closing the TCP connection properly, we would detect this after 2 hours (plus a few minutes). This is of course better than never closing the connection (if KEEP_ALIVE is off), but may not be of much help. So, the first solution would be to lower the timeout value for KEEP_ALIVE. This can only be done for the entire kernel in most operating systems, so if this is lowered to 15 minutes, this will affect all TCP sockets.

2. The second solution is to combine FD_SOCK and FD; the timeout in FD can be set such that it is much lower than the TCP timeout, and this can be configured individually per process. FD_SOCK will already generate a suspect message if the socket was closed abnormally. However, in the case of a crashed switch or host, FD will make sure the socket is eventually closed and the suspect message generated. Example:

```
<FD_SOCK/>
<FD timeout="6000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500"/>
```

This suspects a member when the socket to the neighbor has been closed abonormally (e.g. a process crash, because the OS closes all sockets). However, if a host or switch crashes, then the sockets won't be closed, so, as a second line of defense FD will suspect the neighbor after 30 seconds. Note that with this example, if you have your system stopped in a breakpoint in the debugger, the node you're debugging will be suspected after roughly 30 seconds.

A combination of FD and FD_SOCK provides a solid failure detection layer and for this reason, such technique is used across JGroups configurations included within JBoss Application Server.

10.1.5. Reliable Delivery Protocols

Reliable delivery protocols within the JGroups stack ensure that messages are actually delivered and delivered in the right order (FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments (ACK and NAK). In the ACK mode, the sender resends the message until the acknowledgment is received from the receiver. In the NAK mode, the receiver requests retransmission when it discovers a gap.

10.1.5.1. UNICAST

The UNICAST protocol is used for unicast messages. It uses positive acknowlegements (ACK). It is configured as a sub-element under the JGroups `config` element. If the JGroups stack is configured with the TCP transport protocol, UNICAST is not necessary because TCP itself guarantees FIFO delivery of unicast messages. Here is an example configuration for the `UNICAST` protocol:

```
<UNICAST timeout="300,600,1200,2400,3600"/>
```

There is only one configurable attribute in the `UNICAST` element.

- **timeout** specifies the retransmission timeout (in milliseconds). For instance, if the timeout is "100,200,400,800", the sender resends the message if it hasn't received an ACK after 100 ms the first time, and the second time it waits for 200 ms before resending, and so on. A low value for the first timeout allows for prompt retransmission of dropped messages, but at the potential cost of unnecessary retransmissions if messages aren't actually lost, but rather ACKs just aren't received before the timeout. High values (e.g. "1000,2000,3000") can improve performance if the network has been tuned such that UDP datagram losses are infrequent. High values on lossy networks will hurt performance since later messages will not be delivered until lost messages are retransmitted.

10.1.5.2. NAKACK

The NAKACK protocol is used for multicast messages. It uses negative acknowlegements (NAK). Under this protocol, each message is tagged with a sequence number. The receiver keeps track of the received sequence numbers and delivers the messages in order. When a gap in the series of received sequence numbers is detected, the receiver schedules a task to periodically ask the sender to retransmit the missing message. The task is cancelled if the missing message is received. The NAKACK protocol is configured as the `pbcast.NAKACK` sub-element under the JGroups `config` element. Here is an example configuration.

```
<pbcast.NAKACK max_xmit_size="60000" use_mcast_xmit="false"
  retransmit_timeout="300,600,1200,2400,4800" gc_lag="0"
  discard_delivered_msgs="true"/>
```

The configurable attributes in the `pbcast.NAKACK` element are as follows.

- **retransmit_timeout** specifies the series of timeouts (in milliseconds) after which retransmission is requested if a missing message has not yet been received.
- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just to the node requesting it. This is useful when the *sender*'s network layer tends to drop packets, avoiding the need to individually retransmit to each node.
- **max_xmit_size** specifies the maximum size (in bytes) for a bundled retransmission, if multiple messages are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivered messages on the receiver nodes. By default, nodes save delivered messages so any node can retransmit a lost message in case the original sender has crashed or left the group. However, if we only ask the sender to resend their messages, we can enable this option and discard delivered messages.

- **gc_lag** specifies the number of messages to keep in memory for retransmission even after the periodic cleanup protocol (see *Section 10.1.10, "Distributed Garbage Collection (STABLE)"*) indicates all peers have received the message. Default is `20`.

10.1.6. Group Membership (GMS)

The group membership service (GMS) protocol in the JGroups stack maintains a list of active nodes. It handles the requests to join and leave the cluster. It also handles the SUSPECT messages sent by failure detection protocols. All nodes in the cluster, as well as any interested services like JBoss Cache or HAPartition, are notified if the group membership changes. The group membership service is configured in the `pbcast.GMS` sub-element under the JGroups `config` element. Here is an example configuration.

```
<pbcast.GMS print_local_addr="true"
    join_timeout="3000"
    join_retry_timeout="2000"
    shun="true"
    view_bundling="true"/>
```

The configurable attributes in the `pbcast.GMS` element are as follows.

- **join_timeout** specifies the maximum number of milliseconds to wait for a new node JOIN request to succeed. Retry afterwards.
- **join_retry_timeout** specifies the number of milliseconds to wait after a failed JOIN before trying again.
- **print_local_addr** specifies whether to dump the node's own address to the standard output when starting.
- **shun** specifies whether a node should shun (i.e. disconnect) itself if it receives a cluster view in which it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node becoming the cluster coordinator *during initial connection of the channel. This flag does not prevent a node becoming coordinator later, if the current coordinator leaves the group.*
- **view_bundling** specifies whether multiple JOIN or LEAVE requests arriving at the same time are bundled and handled together at the same time, resulting in only 1 new view incorporating all changes. This is is more efficient than handling each request separately.

10.1.7. Flow Control (FC)

The flow control (FC) protocol tries to adapt the data sending rate to the data receipt rate among nodes. If a sender node is too fast, it might overwhelm the receiver node and result in out-of-

memory conditions or dropped packets that have to be retransmitted. In JGroups, flow control is implemented via a credit-based system. The sender and receiver nodes have the same number of credits (bytes) to start with. The sender subtracts credits by the number of bytes in messages it sends. The receiver accumulates credits for the bytes in the messages it receives. When the sender's credit drops to a threshold, the receivers send some credit to the sender. If the sender's credit is used up, the sender blocks until it receives credits from the receiver. The flow control protocol is configured in the `FC` sub-element under the JGroups `config` element. Here is an example configuration.

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

The configurable attributes in the `FC` element are as follows.

- **max_credits** specifies the maximum number of credits (in bytes). This value should be smaller than the JVM heap size.
- **min_credits** specifies the number of bytes the receipt of which should trigger the receiver to send more credits to the sender.
- **min_threshold** specifies percentage of the `max_credits` that should be used to calculate `min_credits`. Setting this overrides the `min_credits` attribute.
- **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 term "synchronous response" refers to the fact that such an outgoing message is typically a response to an incoming RPC-type message. Not allowing the threads JGroups uses to carry messages up to block in FC is useful in preventing certain deadlock scenarios, so a value of `true` is recommended.

Why is FC needed on top of TCP ? TCP has its own flow control !

The reason is group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. Let's say we have a cluster {A,B,C,D}. D is slow (maybe overloaded), the rest are fast. When A sends a group message, it uses TCP connections A-A (conceptually), A-B, A-C and A-D. So let's say A sends 100 million messages to the cluster. Because TCP's flow control only applies to A-B, A-C and A-D, but not to A-{B,C,D}, where {B,C,D} is the group, it is possible that A, B and C receive the 100M, but D only received 1M messages. (By the way, this is also the reason why we need NAKACK, even though TCP does its own retransmission).

Now JGroups has to buffer all messages in memory for the case when the original sender S dies and a node asks for retransmission of a message sent by S. Because all members buffer all messages they received, they need to purge stable messages (i.e. messages seen by everyone) every now and then. (This is done purging process is managed by the STABLE protocol; see *Section 10.1.10, "Distributed Garbage Collection (STABLE)"*). In the above case, the slow node D will prevent the group from purging messages above 1M, so every member will buffer 99M messages ! This in most cases leads to OOM exceptions. Note that - although the sliding window protocol in TCP will cause writes to block if the window is full - we assume in the above case that this is still much faster for A-B and A-C than for A-D.

So, in summary, even with TCP we need to FC to ensure we send messages at a rate the slowest receiver (D) can handle.

So do I always need FC?

This depends on how the application uses the JGroups channel. Referring to the example above, if there was something about the application that would naturally cause A to slow down its rate of sending because D wasn't keeping up, then FC would not be needed.

A good example of such an application is one that uses JGroups to make synchronous group RPC calls. By synchronous, we mean the thread that makes the call blocks waiting for responses from all the members of the group. In that kind of application, the threads on A that are making calls would block waiting for responses from D, thus naturally slowing the overall rate of calls.

A JBoss Cache cluster configured for REPL_SYNC is a good example of an application that makes synchronous group RPC calls. If a channel is only used for a cache configured for REPL_SYNC, we recommend you remove FC from its protocol stack.

And, of course, if your cluster only consists of two nodes, including FC in a TCP-based protocol stack is unnecessary. There is no group beyond the single peer-to-peer relationship, and TCP's internal flow control will handle that just fine.

Another case where FC may not be needed is for a channel used by a JBoss Cache configured for buddy replication and a single buddy. Such a channel will in many respects act like a two node cluster, where messages are only exchanged with one other node, the buddy. (There may be other messages related to data gravitation that go to all members, but in a properly engineered buddy replication

use case these should be infrequent. But if you remove FC be sure to load test your application.)

10.1.8. Fragmentation (FRAG2)

This protocol fragments messages larger than certain size. Unfragments at the receiver's side. It works for both unicast and multicast messages. It is configured in the FRAG2 sub-element under the JGroups `config` element. Here is an example configuration.

```
<FRAG2 frag_size="60000"/>
```

The configurable attributes in the FRAG2 element are as follows.

- **frag_size** specifies the max frag size in bytes. Messages larger than that are fragmented. For stacks using the UDP transport, this needs to be a value less than 64KB, the maximum UDP datagram size. For TCP-based stacks it needs to be less than the value of max_credits in the FC protocol.

> **Note**
>
> TCP protocol already provides fragmentation but a JGroups fragmentation protocol is still needed if FC is used. The reason for this is that if you send a message larger than FC.max_credits, the FC protocol will block forever. So, frag_size within FRAG2 needs to be set to always be less than FC.max_credits.

10.1.9. State Transfer

The state transfer service requests the application state (serialized as a byte array) from an existing node (i.e., the cluster coordinator) and transfer it to a newly joining node. It tracks the sequence of messages that went into creating the application state, providing a valid starting point for message tracking by reliable delivery protocols like NAKACK and UNICAST. It is configured in the `pbcast.STATE_TRANSFER` sub-element under the JGroups `config` element. It does not have any configurable attribute. Here is an example configuration.

```
<pbcast.STATE_TRANSFER/>
```

10.1.10. Distributed Garbage Collection (STABLE)

In a JGroups cluster, all nodes have to 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. So, the distributed garbage collection service in JGroups periodically purges messages that have seen by all nodes from the memory in each node. The distributed garbage collection service is configured in the `pbcast.STABLE` sub-element under the JGroups `config` element. Here is an example configuration.

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

The configurable attributes in the `pbcast.STABLE` element are as follows.

- **desired_avg_gossip** specifies intervals (in milliseconds) of garbage collection runs. Value `0` disables interval-based execution of garbage collection.
- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Value `0` disables execution of garbage collection based on bytes received.
- **stability_delay** specifies the maximum amount (in milliseconds) of a random delay introduced before a node sends its STABILITY msg at the end of a garbage collection run. The delay gives other nodes concurrently running a STABLE task a change to send first. If used together with max_bytes, this attribute should be set to a small number.

> **Note**
>
> Set the `max_bytes` attribute when you have a high traffic cluster.

10.1.11. Merging (MERGE2)

When a network error occurs (e.g. a crashed switch), the cluster might be partitioned into several different sub-groups. JGroups has "merge" protocols that allow the coordinators in the sub-groups to communicate with each other (once the network heals) and merge back into a single group again. This service is configured in the `MERGE2` sub-element under the JGroups `config` element. Here is an example configuration.

```
<MERGE2 max_interval="100000"
```

```
min_interval="20000"/>
```

The configurable attributes in the `FC` element are as follows.

- **max_interval** specifies the maximum number of milliseconds to wait before sending out a MERGE message.
- **min_interval** specifies the minimum number of milliseconds to wait before send out a MERGE message.

JGroups chooses a random value between `min_interval` and `max_interval` to periodically send out the MERGE message.

> **Note**
>
> The application state maintained by the application using a channel is not merged by JGroups during a merge. This has to be done by the application.

> **Note**
>
> If `MERGE2` is used in conjunction with TCPPING, the `initial_hosts` attribute must contain all the nodes that could potentially be merged back, in order for the merge process to work properly. Otherwise, the merge process may not detect all sub-groups, missing those comprised solely of unlisted members.

10.2. Key JGroups Configuration Tips

10.2.1. Binding JGroups Channels to a Particular Interface

In the Transport Protocols section above, we briefly touched on how the interface to which JGroups will bind sockets is configured. Let's get into this topic in more depth:

First, it's important to understand that the value set in any `bind_addr` element in an XML configuration file will be ignored by JGroups if it finds that system property `jgroups.bind_addr` (or a deprecated earlier name for the same thing, `bind.address`) has been set. The system property trumps XML. If JBoss AS is started with the `-b` (a.k.a. `--host`) switch, the AS will set `jgroups.bind_addr` to the specified value.

Beginning with AS 4.2.0, for security reasons the AS will bind most services to localhost if `-b` is not set. The effect of this is that in most cases users are going to be setting `-b` and thus `jgroups.bind_addr` is going to be set and any XML setting will be ignored.

So, what are *best practices* for managing how JGroups binds to interfaces?

- Binding JGroups to the same interface as other services. Simple, just use `-b`:

```
./run.sh -b 192.168.1.100 -c all
```

- Binding services (e.g., JBoss Web) to one interface, but use a different one for JGroups:

```
./run.sh -b 10.0.0.100 -Djgroups.bind_addr=192.168.1.100 -c all
```

  Specifically setting the system property overrides the `-b` value. This is a common usage pattern; put client traffic on one network, with intra-cluster traffic on another.

- Binding services (e.g., JBoss Web) to all interfaces. This can be done like this:

```
./run.sh -b 0.0.0.0 -c all
```

  However, doing this will not cause JGroups to bind to all interfaces! Instead , JGroups will bind to the machine's default interface. See the Transport Protocols section for how to tell JGroups to receive or send on all interfaces, if that is what you really want.

- Binding services (e.g., JBoss Web) to all interfaces, but specify the JGroups interface:

```
./run.sh -b 0.0.0.0 -Djgroups.bind_addr=192.168.1.100 -c all
```

  Again, specifically setting the system property overrides the `-b` value.

- Using different interfaces for different channels:

```
./run.sh -b 10.0.0.100 -Djgroups.ignore.bind_addr=true -c all
```

This setting tells JGroups to ignore the `jgroups.bind_addr` system property, and instead use whatever is specfied in XML. You would need to edit the various XML configuration files to set the various `bind_addr` attributes to the desired interfaces.

10.2.2. Isolating JGroups Channels

Within JBoss AS, there are a number of services that independently create JGroups channels -- possibly multiple different JBoss Cache services (used for HttpSession replication, EJB3 SFSB replication and EJB3 entity replication), two JBoss Messaging channels, and the general purpose clustering service called HAPartition that underlies most other JBossHA services.

It is critical that these channels only communicate with their intended peers; not with the channels used by other services and not with channels for the same service opened on machines not meant to be part of the group. Nodes improperly communicating with each other is one of the most common issues users have with JBoss AS clustering.

Whom a JGroups channel will communicate with is defined by its group name and, for UDP-based channels, its multicast address and port. So isolating JGroups channels comes down to ensuring

different channels use different values for the group name, the multicast address and, in some cases, the multicast port.

10.2.2.1. Isolating Sets of AS Instances from Each Other

The issue being addressed here is the case where, in the same environment, you have multiple independent clusters running. For example, a production cluster, a staging cluster and a QA cluster. Or multiple clusters in a QA test lab or in a dev team environment. Or a large set of production machines divided into multiple clusters.

To isolate JGroups clusters from other clusters on the network, you need to:

- Make sure the channels in the various clusters use different group names. This is easily to control from the command line arguments used to start JBoss; see *Section 10.2.2.3, "Changing the Group Name"*.
- Make sure the channels in the various clusters use different multicast addresses. This is also easy to control from the command line arguments used to start JBoss; see *Section 10.2.2.4, "Changing the Multicast Address"*.
- If you are not running on Linux, Windows, Solaris or HP-UX, you may also need to ensure that the channels in each cluster use different multicast ports. This is quite a bit more troublesome then using different group names, although it can still be controlled from the command line. See *Section 10.2.2.5, "Changing the Multicast Port"*. Note that using different ports should not be necessary if your servers are running on Linux, Windows, Solaris or HP-UX.

10.2.2.2. Isolating Channels for Different Services on the Same Set of AS Instances

The issue being addressed here is the normal case where we have a cluster of 3 machines, each of which has, for example, an HAPartition deployed along with a JBoss Cache used for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. Ensuring proper isolation of these channels is straightforward, and generally speaking the AS handles it for you without any special effort on your part. So most readers can skip this section.

To isolate JGroups channels for different services on the same set of AS instances from each other, each channel must have its own group name. The configurations that ship with JBoss AS of course ensure that this is the case. If you create a custom service that directly uses JGroups, just make sure you use a unique group name. If you create a custom JBoss Cache configuration, make sure you provide a unique value in the `clusterName` configuration property.

In releases prior to AS 5, different channels running in the same AS also had to use unique multicast ports. With the JGroups shared transport introduced in AS 5 (see *Section 3.1.2, "The JGroups Shared Transport"*), it is now common for multiple channels to use the same tranpsort protocol and its sockets. This makes configuration easier, which is one of the main benefits of the shared transport. However, if you decide to create your own custom JGroups protocol stack

configuration, be sure to configure its transport protocols with a multicast port that is different from the ports used in other protocol stacks.

10.2.2.3. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. For all the standard clustered services, we make it easy for you to create unique groups names by simply using the `-g` (a.k.a. `--partition`) switch when starting JBoss:

```
./run.sh -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the `jboss.partition.name` system property, which is used as a component in the configuration of the group name in all the standard clustering configuration files. For example,

```
<property name="clusterName">${jboss.partition.name:DefaultPartition}-SFSBCache</property>
```

10.2.2.4. Changing the Multicast Address

The `-u` (a.k.a. `--udp`) command line switch may be used to control the multicast address used by the JGroups channels opened by all standard AS services.

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all
```

This switch sets the `jboss.partition.udpGroup` system property, which you can see referenced in all of the standard protocol stack configurations in JBoss AS:

```
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}" ....
```

> **Why isn't it sufficient to change the group name?**
>
> If channels with different group names share the same multicast address and port, the lower level JGroups protocols in each channel will see, process and eventually discard messages intended for the other group. This will at a minimum hurt performance and can lead to anomalous behavior.

10.2.2.5. Changing the Multicast Port

On some operating systems (Mac OS X for example), using different `-g` and `-u` values isn't sufficient to isolate clusters; the channels running in the different clusters need to use different multicast ports. Unfortunately, setting the multicast ports is not quite as simple as `-g` and `-u`. By default, a JBoss AS instance running the all configuration will use up to two different instances of

the JGroups UDP transport protocol, and will thus open two multicast sockets. You can control the ports those sockets use by using system properties on the command line. For example,

```
/run.sh -u 230.1.2.3 -g QAPartition -b 192.168.1.100 -c all \\
      -Djboss.jgroups.udp.mcast_port=12345 -Djboss.messaging.datachanneludpport=23456
```

The `jboss.messaging.datachanneludpport` property controls the multicast port used by the MPING protocol in JBoss Messaging's DATA channel. The `jboss.jgroups.udp.mcast_port` property controls the multicast port used by the UDP transport protocol shared by all other clustered services.

The set of JGroups protocol stack configurations included in the `$JBOSS_HOME/server/all/cluster/jgroups-channelfactory.sar/META-INF/jgroups-channelfactory-stacks.xml` file includes a number of other example protocol stack configurations that the standard AS distribution doesn't actually use. Those configurations also use system properties to set any multicast ports. So, if you reconfigure some AS service to use one of those protocol stack configurations, just use the appropriate system property to control the port from the command line.

Why do I need to change the multicast port if I change the address?

It should be sufficient to just change the address, but unfortunately the handling of multicast sockets is one area where the JVM fails to hide OS behavior differences from the application. The `java.net.MulticastSocket` class provides different overloaded constructors. On some operating systems, if you use one constructor variant, there is a problem whereby packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address on which they are listening. We refer to this as the "promiscuous traffic" problem. On most operating systems that exhibit the promiscuous traffic problem (i.e. Linux, Solaris and HP-UX) JGroups can use a different constructor variant that avoids the problem. However, on some OSs with the promiscuous traffic problem (e.g. Mac OS X), multicast does not work properly if the other constructor variant is used. So, on these operating systems the recommendation is to configure different multicast ports for different clusters.

10.2.3. Improving UDP Performance by Configuring OS UDP Buffer Limits

By default, the JGroups channels in JBoss AS use the UDP transport protocol in order to take advantage of IP multicast. However, one disadvantage of UDP is it does not come with the

reliable delivery guarantees provided by TCP. The protocols discussed in *Section 10.1.5, "Reliable Delivery Protocols"* allow JGroups to guarantee delivery of UDP messages, but those protocols are implemented in Java, not at the OS network layer. To get peak performance from a UDP-based JGroups channel it is important to limit the need for JGroups to retransmit messages by limiting UDP datagram loss.

One of the most common causes of lost UDP datagrams is an undersized receive buffer on the socket. The UDP protocol's `mcast_recv_buf_size` and `ucast_recv_buf_size` configuration attributes are used to specify the amount of receive buffer JGroups *requests* from the OS, but the actual size of the buffer the OS will provide is limited by OS-level maximums. These maximums are often very low:

Table 10.1. Default Max UDP Buffer Sizes

Operating System	Default Max UDP Buffer (in bytes)
Linux	131071
Windows	No known limit
Solaris	262144
FreeBSD, Darwin	262144
AIX	1048576

The command used to increase the above limits is OS-specific. The table below shows the command required to increase the maximum buffer to 25MB. In all cases root privileges are required:

Table 10.2. Commands to Change Max UDP Buffer Sizes

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

10.3. JGroups Troubleshooting

10.3.1. Nodes do not form a cluster

Make sure your machine is set up correctly for IP multicast. There are 2 test programs that can be used to detect this: McastReceiverTest and McastSenderTest. Go to the `$JBOSS_HOME/server/all/lib` directory and start McastReceiverTest, for example:

```
java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -mcast_addr
 224.10.10.10 -port 5555
```

Then in another window start `McastSenderTest`:

```
java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr
 224.10.10.10 -port 5555
```

If you want to bind to a specific network interface card (NIC), use `-bind_addr 192.168.0.2`, where 192.168.0.2 is the IP address of the NIC to which you want to bind. Use this parameter in both the sender and the receiver.

You should be able to type in the `McastSenderTest` window and see the output in the `McastReceiverTest` window. If not, try to use -ttl 32 in the sender. If this still fails, consult a system administrator to help you setup IP multicast correctly, and ask the admin to make sure that multicast will work on the interface you have chosen or, if the machines have multiple interfaces, ask to be told the correct interface. Once you know multicast is working properly on each machine in your cluster, you can repeat the above test to test the network, putting the sender on one machine and the receiver on another.

10.3.2. Causes of missing heartbeats in FD

Sometimes a member is suspected by FD because a heartbeat ack has not been received for some time T (defined by timeout and max_tries). This can have multiple reasons, e.g. in a cluster of A,B,C,D; C can be suspected if (note that A pings B, B pings C, C pings D and D pings A):

- B or C are running at 100% CPU for more than T seconds. So even if C sends a heartbeat ack to B, B may not be able to process it because it is at 100%
- B or C are garbage collecting, same as above.
- A combination of the 2 cases above
- The network loses packets. This usually happens when there is a lot of traffic on the network, and the switch starts dropping packets (usually broadcasts first, then IP multicasts, TCP packets last).
- B or C are processing a callback. Let's say C received a remote method call over its channel and takes T+1 seconds to process it. During this time, C will not process any other messages, including heartbeats, and therefore B will not receive the heartbeat ack and will suspect C.

JBoss Cache Configuration and Deployment

JBoss Cache provides the underlying distributed caching support used by many of the standard clustered services in a JBoss AS cluster. You can also deploy JBoss Cache in your own application to handle custom caching requirements. In this chapter we provide some background on the main configuration options available with JBoss Cache, with an emphasis on how those options relate to the JBoss Cache usage by the standard clustered services the AS provides. We then discuss the different options available for deploying a custom cache in the AS.

Users considering deploying JBoss Cache for direct use by their own application are strongly encouraged to read the JBoss Cache documentation available at http://www.jboss.org/jbosscache.

See also *Section 3.2, "Distributed Caching with JBoss Cache"* for information on how the standard JBoss AS clustered services use JBoss Cache.

11.1. Key JBoss Cache Configuration Options

JBoss AS ships with a reasonable set of default JBoss Cache configurations that are suitable for the standard clustered service use cases (e.g. web session replication or JPA/Hibernate caching). Most applications that involve the standard clustered services just work out of the box with the default configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements. In this section we provide a brief overview of some of the key configuration choices. This is by no means a complete discussion; for full details users interested in moving beyond the default configurations are encouraged to read the JBoss Cache documentation available at http://www.jboss.org/jbosscache.

Most JBoss Cache configuration examples in this section use the JBoss Microcontainer schema for building up an `org.jboss.cache.config.Configuration` object graph from XML. JBoss Cache has its own custom XML schema, but the standard JBoss AS CacheManager service uses the JBoss Microcontainer schema to be consistent with most other internal AS services.

Before getting into the key configuration options, let's have a look at the most likely place that a user would encounter them.

11.1.1. Editing the CacheManager Configuration

As discussed in *Section 3.2.1, "The JBoss AS CacheManager Service"*, the standard JBoss AS clustered services use the CacheManager service as a factory for JBoss Cache instances. So, cache configuration changes are likely to involve edits to the CacheManager service.

> **Note**
>
> Users can also use the CacheManager as a factory for custom caches used by directly by their own applications; see *Section 11.2.1, "Deployment Via the CacheManager Service"*.

The CacheManager is configured via the `deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml` file. The element most likely to be edited is the "CacheConfigurationRegistry" bean, which maintains a registry of all the named JBC configurations the CacheManager knows about. Most edits to this file would involve adding a new JBoss Cache configuration or changing a property of an existing one.

The following is a redacted version of the "CacheConfigurationRegistry" bean configuration:

```
<bean name="CacheConfigurationRegistry"
    class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">

  <!-- If users wish to add configs using a more familiar JBC config format
       they can add them to a cache-configs.xml file specified by this property.
       However, use of the microcontainer format used below is recommended.
  <property name="configResource">META-INF/jboss-cache-configs.xml</property>
  -->

  <!-- The configurations. A Map<String name, Configuration config> -->
  <property name="newConfigurations">
    <map keyClass="java.lang.String" valueClass="org.jboss.cache.config.Configuration">

  <!-- The standard configurations follow.  You can add your own and/or edit these. -->

  <!-- Standard cache used for web sessions -->
  <entry><key>standard-session-cache</key>
  <value>
    <bean name="StandardSessionCacheConfig" class="org.jboss.cache.config.Configuration">


        .... details of the standard-session-cache configuration
    </bean>
  </value>
  </entry>

  <!-- Appropriate for web sessions with FIELD granularity -->
  <entry><key>field-granularity-session-cache</key>
  <value>
```

```
     <bean name="FieldSessionCacheConfig" class="org.jboss.cache.config.Configuration">

        .... details of the field-granularity-standard-session-cache configuration
     </bean>

  </value>

  </entry>

  ... entry elements for the other configurations

 </map>
 </property>
</bean>
```

The actual JBoss Cache configurations are specified using the JBoss Microcontainer's schema rather than one of the standard JBoss Cache configuration formats. When JBoss Cache parses one of its standard configuration formats, it creates a Java Bean of type `org.jboss.cache.config.Configuration` with a tree of child Java Beans for some of the more complex sub-configurations (i.e. cache loading, eviction, buddy replication). Rather than delegating this task of XML parsing/Java Bean creation to JBC, we let the AS's microcontainer do it directly. This has the advantage of making the microcontainer aware of the configuration beans, which in later AS 5.x releases will be helpful in allowing external management tools to manage the JBC configurations.

The configuration format should be fairly self-explanatory if you look at the standard configurations the AS ships; they include all the major elements. The types and properties of the various java beans that make up a JBoss Cache configuration can be seen in the JBoss Cache javadocs. Here is a fairly complete example:

```
<bean name="StandardSFSBCacheConfig" class="org.jboss.cache.config.Configuration">

  <!--  No transaction manager lookup -->

  <!-- Name of cluster. Needs to be the same for all members -->
  <property name="clusterName">
    ${jboss.partition.name:DefaultPartition}-SFSBCache
  </property>
  <!-- Use a UDP (multicast) based stack. Need JGroups flow control (FC)
       because we are using asynchronous replication. -->
  <property name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
  <property name="fetchInMemoryState">true</property>
```

```
<property name="nodeLockingScheme">PESSIMISTIC</property>
<property name="isolationLevel">REPEATABLE_READ</property>
<property name="cacheMode">REPL_ASYNC</property>

<property name="useLockStriping">false</property>

<!-- Number of milliseconds to wait until all responses for a
     synchronous call have been received. Make this longer
     than lockAcquisitionTimeout.-->
<property name="syncReplTimeout">17500</property>
<!-- Max number of milliseconds to wait for a lock acquisition -->
<property name="lockAcquisitionTimeout">15000</property>
<!-- The max amount of time (in milliseconds) we wait until the
 state (ie. the contents of the cache) are retrieved from
 existing members at startup. -->
<property name="stateRetrievalTimeout">60000</property>

<!--
 SFSBs use region-based marshalling to provide for partial state
 transfer during deployment/undeployment.
-->
<property name="useRegionBasedMarshalling">false</property>
<!-- Must match the value of "useRegionBasedMarshalling" -->
<property name="inactiveOnStartup">false</property>

<!-- Disable asynchronous RPC marshalling/sending -->
<property name="serializationExecutorPoolSize">0</property>
<!-- We have no asynchronous notification listeners -->
<property name="listenerAsyncPoolSize">0</property>

<property name="exposeManagementStatistics">true</property>

<property name="buddyReplicationConfig">
  <bean class="org.jboss.cache.config.BuddyReplicationConfig">

    <!--  Just set to true to turn on buddy replication -->
    <property name="enabled">false</property>

    <!-- A way to specify a preferred replication group.  We try
         and pick a buddy who shares the same pool name (falling
         back to other buddies if not available). -->
    <property name="buddyPoolName">default</property>
```

```
      <property name="buddyCommunicationTimeout">17500</property>

      <!-- Do not change these -->
      <property name="autoDataGravitation">false</property>
      <property name="dataGravitationRemoveOnFind">true</property>
      <property name="dataGravitationSearchBackupTrees">true</property>

      <property name="buddyLocatorConfig">
        <bean class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
          <!-- The number of backup nodes we maintain -->
          <property name="numBuddies">1</property>
          <!-- Means that each node will *try* to select a buddy on
               a different physical host. If not able to do so
               though, it will fall back to colocated nodes. -->
          <property name="ignoreColocatedBuddies">true</property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="cacheLoaderConfig">
    <bean class="org.jboss.cache.config.CacheLoaderConfig">
          <!-- Do not change these -->
          <property name="passivation">true</property>
          <property name="shared">false</property>

          <property name="individualCacheLoaderConfigs">
            <list>
              <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
                <!-- Where passivated sessions are stored -->
                <property name="location">${jboss.server.data.dir}${/}sfsb</property>
                <!-- Do not change these -->
                <property name="async">false</property>
                <property name="fetchPersistentState">true</property>
                <property name="purgeOnStartup">true</property>
                <property name="ignoreModifications">false</property>
                <property name="checkCharacterPortability">false</property>
              </bean>
            </list>
          </property>
    </bean>
  </property>

  <!-- EJBs use JBoss Cache eviction -->
  <property name="evictionConfig">
```

```
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeupInterval">5000</property>
      <!--  Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName">/</property>
          <property name="evictionAlgorithmConfig">
            <bean class="org.jboss.cache.eviction.NullEvictionAlgorithmConfig"/>
          </property>
        </bean>
      </property>
      <!-- EJB3 integration code will programatically create
           other regions as beans are deployed -->
    </bean>
  </property>
</bean>
```

Basically, the XML specifies the creation of an `org.jboss.cache.config.Configuration` java bean and the setting of a number of properties on that bean. Most of the properties are of simple types, but some, such as `buddyReplicationConfig` and `cacheLoaderConfig` take various types java beans as their values.

Next we'll look at some of the key configuration options.

11.1.2. Cache Mode

JBoss Cache's `cacheMode` configuration attribute combines into a single property two related aspects:

Handling of Cluster Updates

This controls how a cache instance on one node should notify the rest of the cluster when it makes changes in its local state. There are three options:

- **Synchronous** means the cache instance sends a message to its peers notifying them of the change(s) and before returning waits for them to acknowledge that they have applied the same changes. If the changes are made as part of a JTA transaction, this is done as part of a 2 phase-commit process during transaction commit. Any locks are held until this acknowledgment is received. Waiting for acknowledgement from all nodes adds delays, but it ensures consistency around the cluster. Synchronous mode is needed when all the nodes in the cluster may access the cached data resulting in a high need for consistency.
- **Asynchronous** means the cache instance sends a message to its peers notifying them of the change(s) and then immediately returns, without any acknowledgement that they have applied the same changes. It *does not* mean sending the message is handled by some other thread besides the one that changed the cache content; the thread that makes the change

still spends some time dealing with sending messages to the cluster, just not as much as with synchronous communication. Asynchronous mode is most useful for cases like session replication, where the cache doing the sending expects to be the only one that accesses the data and the cluster messages are used to provide backup copies in case of failure of the sending node. Asynchronous messaging adds a small risk that a later user request that fails over to another node may see out-of-date state, but for many session-type applications this risk is acceptable given the major performance benefits asynchronous mode has over synchronous mode.

- **Local** means the cache instance doesn't send a message at all. A JGroups channel isn't even used by the cache. JBoss Cache has many useful features besides its clustering capabilities and is a very useful caching library even when not used in a cluster. Also, even in a cluster, some cached data does not need to be kept consistent around the cluster, in which case Local mode will improve performance. Caching of JPA/Hibernate query result sets is an example of this; Hibernate's second level caching logic uses a separate mechanism to invalidate stale query result sets from the second level cache, so JBoss Cache doesn't need to send messages around the cluster for a query result set cache.

Replication vs. Invalidation

This aspect deals with the content of messages sent around the cluster when a cache changes its local state, i.e. what should the other caches in the cluster do to reflect the change:

- **Replication** means the other nodes should update their state to reflect the new state on the sending node. 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 the other nodes should remove the changed state from their local state. 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. However, it is only an option if the removed state can be retrieved from another source. It is an excellent option for a clustered JPA/Hibernate entity cache, since the cached state can be re-read from the database.

These two aspects combine to form 5 valid values for the `cacheMode` configuration attribute:

- **LOCAL** means no cluster messages are needed.
- **REPL_SYNC** means synchronous replication messages are sent.
- **REPL_ASYNC** means asynchronous replication messages are sent.
- **INVALIDATION_SYNC** means synchronous invalidation messages are sent.
- **INVALIDATION_ASYNC** means asynchronous invalidation messages are sent.

11.1.3. Transaction Handling

JBoss Cache integrates with JTA transaction managers to allow transactional access to the cache. When JBoss Cache detects the presence of a transaction, any locks are held for the life of the

transaction, changes made to the cache will be reverted if the transaction rolls back, and any cluster-wide messages sent to inform other nodes of changes are deferred and sent in a batch as part of transaction commit (reducing chattiness).

Integration with a transaction manager is accomplished by setting the `transactionManagerLookupClass` configuration attribute; this specifies the fully qualified class name of a class JBoss Cache can use to find the local transaction manager. Inside JBoss AS, this attribute would have one of two values:

- **org.jboss.cache.transaction.JBossTransactionManagerLookup**

  This finds the standard transaction manager running in the application server. Use this for any custom caches you deploy where you want caching to participate in any JTA transactions.

- **org.jboss.cache.transaction.BatchModeTransactionManagerLookup**

  This is used in the cache configurations used for web session and EJB SFSB caching. It specifies a simple mock `TransactionManager` that ships with JBoss Cache called the `BatchModeTransactionManager`. This transaction manager is not a true JTA transaction manager and should not be used for anything other than JBoss Cache. Its usage in JBoss AS is to get most of the benefits of JBoss Cache's transactional behavior for the session replication use cases, but without getting tangled up with end user transactions that may run during a request.

> **Note**
>
> For caches used for JPA/Hibernate caching, the `transactionManagerLookupClass` should not be configured. Hibernate internally configures the cache to use the same transaction manager it is using for database access.

11.1.4. Concurrent Access

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`:

- **MVCC** or multi-versioned concurrency control, is a locking scheme commonly used by modern database implementations to control fast, safe concurrent access to shared data. JBoss Cache 3.x uses an innovative implementation of MVCC as the default locking scheme. MVCC is designed to provide the following features for concurrent access:
  - Readers that don't block writers
  - Writers that fail fast

It achieves this by using data versioning and copying for concurrent writers. The theory is that readers continue reading shared state, while writers copy the shared state, increment a version id, and write that shared state back after verifying that the version is still valid (i.e., another concurrent writer has not changed this state first).

MVCC is the recommended choice for JPA/Hibernate entity caching.

- **PESSIMISTIC** locking involves threads/transactions acquiring either exclusive or non-exclusive locks on nodes before reading or writing. Which is acquired depends on the `isolationLevel` (see below) 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 considerably more overhead than MVCC 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). A write will also be delayed due to ongoing reads.

  Generally MVCC is a better choice than PESSIMISTIC, which is deprecated as of JBoss Cache 3.0. But, for the session caching usage in JBoss AS 5.0.0, PESSIMISTIC is still the default. This is largely because 1) for the session use case there are generally not concurrent threads accessing the same cache location, so the benefits of MVCC are not as great, and 2) the AS development team wanted to continue to evaluate MVCC in the session use case before moving to it as the default.

- **OPTIMISTIC** locking seeks to improve upon the concurrency available with PESSIMISTIC by creating a "workspace" for each request/transaction that accesses the cache. Data accessed by the request/transaction (even reads) is *copied* into the workspace, which is adds overhead. 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 are inconsistencies. Otherwise changes to the workspace are applied to the main cache.

  OPTIMISTIC locking is deprecated but is still provided to support backward compatibility. Users are encouraged to use MVCC instead, which provides the same benefits at lower cost.

The `isolationLevel` attribute has two possible values **READ_COMMITTED** and **REPEATABLE_READ** which correspond in semantic to database-style isolation levels. Previous versions of JBoss Cache supported all 5 database isolation levels, and if an unsupported isolation level is configured, it is either upgraded or downgraded to the closest supported level.

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.

11.1.5. JGroups Integration

Each JBoss Cache instance internally uses a JGroups `Channel` to handle group communications. Inside JBoss AS, we strongly recommend that you use the AS's JGroups Channel Factory service (see *Section 3.1.1, "The Channel Factory Service"*) as the source for your cache's `Channel`. In

this section we discuss how to configure your cache to get it's channel from the Channel Factory; if you wish to configure the channel in some other way see the JBoss Cache documentation.

Caches obtained from the CacheManager Service

This is the simplest approach. The CacheManager service already has a reference to the Channel Factory service, so the only configuration task is to configure the name of the JGroups protocol stack configuration to use.

If you are configuring your cache via the CacheManager service's `jboss-cache-manager-jboss-beans.xml` file (see *Section 11.2.1, "Deployment Via the CacheManager Service"*), add the following to your cache configuration, where the value is the name of the protocol stack configuration.:

```
<property name="multiplexerStack">udp</property>
```

Caches Deployed via a `-jboss-beans.xml` File

If you are deploying a cache via a JBoss Microcontainer `-jboss-beans.xml` file (see *Section 11.2.3, "Deployment Via a `-jboss-beans.xml` File"*), you need inject a reference to the Channel Factory service as well as specifying the protocol stack configuration:

```
<property name="runtimeConfig">
  <bean class="org.jboss.cache.config.RuntimeConfig">
    <property name="muxChannelFactory"><inject bean="JChannelFactory"/></property>
  </bean>
</property>
<property name="multiplexerStack">udp</property>
```

Caches Deployed via a `-service.xml` File

If you are deploying a cache MBean via `-service.xml` file (see *Section 11.2.2, "Deployment Via a `-service.xml` File"*), `CacheJmxWrapper` is the class of your MBean; that class exposes a `MuxChannelFactory` MBean attribute. You dependency inject the Channel Factory service into this attribute, and set the protocol stack name via the `MultiplexerStack` attribute:

```
<attribute name="MuxChannelFactory"><inject bean="JChannelFactory"/></attribute>
<attribute name="MultiplexerStack">udp</attribute>
```

11.1.6. Eviction

Eviction allows the cache to control memory by removing data (typically the least frequently used data). If you wish to configure eviction for a custom cache, see the JBoss Cache documentation

for all of the available options. For details on configuring it for JPA/Hibernate caching, see the Eviction chapter in the "Using JBoss Cache as a Hibernate Second Level Cache" guide at http://www.jboss.org/jbossclustering/docs/hibernate-jbosscache-guide-3.pdf. For web session caches, eviction should not be configured; the distributable session manager handles eviction itself. For EJB 3 SFSB caches, stick with the eviction configuration in the AS's standard `sfsb-cache` configuration (see *Section 3.2.1, "The JBoss AS CacheManager Service"*). The EJB container will configure eviction itself using the values included in each bean's configuration.

11.1.7. Cache Loaders

Cache loading allows JBoss Cache to store data in a persistent store in addition to what it keeps in memory. This data can either be an overflow, where the data in the persistent store is not reflected in memory. Or it can be a superset 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. Which of these two modes is used depends on the setting of the `passivation` flag in the JBoss Cache cache loader configuration section. A `true` value means the persistent store acts as an overflow area written to when data is evicted from the in-memory cache.

If you wish to configure cache loading for a custom cache, see the JBoss Cache documentation for all of the available options. Do not configure cache loading for a JPA/Hibernate cache, as the database itself serves as a persistent store; adding a cache loader is just redundant.

The caches used for web session and EJB3 SFSB caching use passivation. Next we'll discuss the cache loader configuration for those caches in some detail.

11.1.7.1. CacheLoader Configuration for Web Session and SFSB Caches

HttpSession and SFSB passivation rely on JBoss Cache's Cache Loader passivation for storing and retrieving the passivated sessions. Therefore the cache instance used by your webapp's clustered session manager or your bean's EJB container must be configured to enable Cache Loader passivaton.

In most cases you don't need to do anything to alter the cache loader configurations for the standard web session and SFSB caches; the standard JBoss AS configurations should suit your needs. The following is a bit more detail in case you're interested or want to change from the defaults.

The Cache Loader configuration for the `standard-session-cache` config serves as a good example:

```
<property name="cacheLoaderConfig">
  <bean class="org.jboss.cache.config.CacheLoaderConfig">
        <!-- Do not change these -->
        <property name="passivation">true</property>
```

```
        <property name="shared">false</property>

        <property name="individualCacheLoaderConfigs">
          <list>
            <bean class="org.jboss.cache.loader.FileCacheLoaderConfig">
              <!-- Where passivated sessions are stored -->
              <property name="location">${jboss.server.data.dir}${/}field-session</property>
              <!-- Do not change these -->
              <property name="async">false</property>
              <property name="fetchPersistentState">true</property>
              <property name="purgeOnStartup">true</property>
              <property name="ignoreModifications">false</property>
              <property name="checkCharacterPortability">false</property>
            </bean>
          </list>
        </property>
    </bean>
</property>
```

Some explanation:

- **passivation** property MUST be `true`
- **shared** property MUST be `false`. Do not passivate sessions to a shared persistent store, otherwise if another node activates the session, it will be gone from the persistent store and also gone from memory on other nodes that have passivated it. Backup copies will be lost.
- **individualCacheLoaderConfigs** property accepts a list of Cache Loader configurations. JBC allows you to chain cache loaders; see the JBoss Cache docs. For the session passivation use case a single cache loader is sufficient.
- **class** attribute on a cache loader config bean must refer to the configuration class for a cache loader implementation (e.g. `org.jboss.cache.loader.FileCacheLoaderConfig` or `org.jboss.cache.loader.JDBCCacheLoaderConfig`). See the JBoss Cache documentation for more on the available CacheLoader implementations. If you wish to use JDBCCacheLoader (to persist to a database rather than the filesystem used by FileCacheLoader) note the comment above about the `shared` property. Don't use a shared database, or at least not a shared table in the database. Each node in the cluster must have its own storage location.
- **location** property for FileCacheLoaderConfig defines the root node of the filesystem tree where passivated sessions should be stored. The default is to store them in your JBoss AS configuration's `data` directory.
- **async** MUST be `false` to ensure passivated sessions are promptly written to the persistent store.

- **fetchPersistentState** property MUST be `true` to ensure passivated sessions are included in the set of session backup copies transferred over from other nodes when the cache starts.
- **purgeOnStartup** should be `true` to ensure out-of-date session data left over from a previous shutdown of a server doesn't pollute the current data set.
- **ignoreModifications** should be `false`
- **checkCharacterPortability** should be `false` as a minor performance optimization.

11.1.8. Buddy Replication

Buddy Replication is a JBoss Cache feature that allows you to suppress replicating your data to all instances in a cluster. Instead, each instance picks one or more 'buddies' in the cluster, and only replicates to those specific buddies. This greatly helps scalability as there is no longer a memory and network traffic impact every time another instance is added to a cluster.

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; any node can handle a request for any data. However, data gravitation is expensive and should not be a frequent occurence; 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. This makes buddy replication primarily useful for session-type applications with session affinity (a.k.a. "sticky sessions") where all requests for a particular session are normally handled by a single server.

Buddy replication can be enabled for the web session and EJB3 SFSB caches. Do not add buddy replication to the cache configurations used for other standard clustering services (e.g. JPA/Hibernate caching). Services not specifically engineered for buddy replication are highly unlikely to work correctly if it is introduced.

Configuring buddy replication is fairly straightforward. As an example we'll look at the buddy replication configuration section from the CacheManager service's `standard-session-cache` config:

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

    <!--  Just set to true to turn on buddy replication -->
    <property name="enabled">true</property>

    <!-- A way to specify a preferred replication group.  We try
         and pick a buddy who shares the same pool name (falling
         back to other buddies if not available). -->
```

```
    <property name="buddyPoolName">default</property>

    <property name="buddyCommunicationTimeout">17500</property>

    <!-- Do not change these -->
    <property name="autoDataGravitation">false</property>
    <property name="dataGravitationRemoveOnFind">true</property>
    <property name="dataGravitationSearchBackupTrees">true</property>

    <property name="buddyLocatorConfig">
      <bean class="org.jboss.cache.buddyreplication.NextMemberBuddyLocatorConfig">
        <!-- The number of backup copies we maintain -->
        <property name="numBuddies">1</property>
        <!-- Means that each node will *try* to select a buddy on
             a different physical host. If not able to do so
             though, it will fall back to colocated nodes. -->
        <property name="ignoreColocatedBuddies">true</property>
       </bean>
    </property>
  </bean>
</property>
```

The main things you would be likely to configure are:

- **buddyReplicationEnabled** -- `true` if you want buddy replication; `false` if data should be replicated to all nodes in the cluster, in which case none of the other buddy replication configurations matter.
- **numBuddies** -- to how many backup nodes should each node replicate its state.
- **buddyPoolName** -- allows logical subgrouping of nodes within the cluster; if possible, buddies will be chosen from nodes in the same buddy pool.

The `ignoreColocatedBuddies` switch means that when the cache is trying to find a buddy, it will if possible not choose a buddy on the same physical host as itself. If the only server it can find is running on its own machine, it will use that server as a buddy.

Do not change the settings for `autoDataGravitation`, `dataGravitationRemoveOnFind` and `dataGravitationSearchBackupTrees`. Session replication will not work properly if these are changed.

11.2. Deploying Your Own JBoss Cache Instance

It's quite common for users to deploy their own instances of JBoss Cache inside JBoss AS for custom use by their applications. In this section we describe the various ways caches can be deployed.

11.2.1. Deployment Via the CacheManager Service

The standard JBoss clustered services that use JBoss Cache obtain a reference to their cache from the AS's CacheManager service (see *Section 3.2.1, "The JBoss AS CacheManager Service"*). End user applications can do the same thing; here's how.

Section 11.1.1, "Editing the CacheManager Configuration" shows the configuration of the CacheManager's "CacheConfigurationRegistry" bean. To add a new configuration, you would add an additional element inside that bean's `newConfigurations` <map>:

```
<bean name="CacheConfigurationRegistry"
    class="org.jboss.ha.cachemanager.DependencyInjectedConfigurationRegistry">
  .....
  <property name="newConfigurations">
   <map keyClass="java.lang.String" valueClass="org.jboss.cache.config.Configuration">

    <entry><key>my-custom-cache</key>
     <value>
        <bean name="MyCustomCacheConfig" class="org.jboss.cache.config.Configuration">

        .... details of the my-custom-cache configuration
       </bean>
     </value>
    </entry>
    .....
```

See *Section 11.1.1, "Editing the CacheManager Configuration"* for an example configuration.

11.2.1.1. Accessing the CacheManager

Once you've added your cache configuration to the CacheManager, the next step is to provide a reference to the CacheManager to your application. There are three ways to do this:

- **Dependency Injection**

  If your application uses the JBoss Microcontainer for configuration, the simplest mechanism is to have it inject the CacheManager into your service.

```
<bean name="MyService" class="com.example.MyService">
  <property name="cacheManager"><inject bean="CacheManager"/></property>
</bean>
```

- **JNDI Lookup**

Alternatively, you can find look up the CacheManger is JNDI. It is bound under `java:CacheManager`.

```
import org.jboss.ha.cachemanager.CacheManager;

public class MyService {
  private CacheManager cacheManager;

  public void start() throws Exception {
     Context ctx = new InitialContext();
     cacheManager = (CacheManager) ctx.lookup("java:CacheManager");
  }
}
```

- **CacheManagerLocator**

  JBoss AS also provides a service locator object that can be used to access the CacheManager.

```
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
  private CacheManager cacheManager;

  public void start() throws Exception {
     CacheManagerLocator locator = CacheManagerLocator.getCacheManagerLocator();
     // Locator accepts as param a set of JNDI properties to help in lookup;
     // this isn't necessary inside the AS
     cacheManager = locator.getCacheManager(null);
  }
}
```

Once a reference to the CacheManager is obtained; usage is simple. Access a cache by passing in the name of the desired configuration. The CacheManager will not start the cache; this is the responsibility of the application. The cache may, however, have been started by another application running in the cache server; the cache may be shared. When the application is done using the cache, it should not stop. Just inform the CacheManager that the cache is no longer being used; the manager will stop the cache when all callers that have asked for the cache have released it.

```
import org.jboss.cache.Cache;
```

```
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
   private CacheManager cacheManager;
   private Cache cache;

   public void start() throws Exception {
      Context ctx = new InitialContext();
      cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

      // "true" param tells the manager to instantiate the cache if
      // it doesn't exist yet
      cache = cacheManager.getCache("my-cache-config", true);

      cache.start();
   }

   public void stop() throws Exception {
      cacheManager.releaseCache("my-cache-config");
   }
}
```

The CacheManager can also be used to access instances of POJO Cache.

```
import org.jboss.cache.pojo.PojoCache;
import org.jboss.ha.cachemanager.CacheManager;
import org.jboss.ha.framework.server.CacheManagerLocator;

public class MyService {
   private CacheManager cacheManager;
   private PojoCache pojoCache;

   public void start() throws Exception {
      Context ctx = new InitialContext();
      cacheManager = (CacheManager) ctx.lookup("java:CacheManager");

      // "true" param tells the manager to instantiate the cache if
      // it doesn't exist yet
      pojoCache = cacheManager.getPojoCache("my-cache-config", true);

      pojoCache.start();
   }
```

```
  public void stop() throws Exception {
     cacheManager.releasePojoCache("my-cache-config");
  }
}
```

11.2.2. Deployment Via a `-service.xml` File

As in JBoss 4, you can also deploy a JBoss Cache instance as an MBean service via a `-service.xml` file. The primary difference from JBoss 4 is the value of the `code` attribute in the `mbean` element. In JBoss 4, this was `org.jboss.cache.TreeCache`; in JBoss 5 it is `org.jboss.cache.jmx.CacheJmxWrapper`. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jboss.cache.jmx.CacheJmxWrapper"
         name="foo:service=ExampleCacheJmxWrapper">

    <attribute name="TransactionManagerLookupClass">
      org.jboss.cache.transaction.JBossTransactionManagerLookup
    </attribute>

    <attribute name="MuxChannelFactory"><inject bean="JChannelFactory"/></attribute>

    <attribute name="MultiplexerStack">udp</attribute>
    <attribute name="ClusterName">Example-EntityCache</attribute>
    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
    <attribute name="CacheMode">REPL_SYNC</attribute>
    <attribute name="InitialStateRetrievalTimeout">15000</attribute>
    <attribute name="SyncReplTimeout">20000</attribute>
    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="ExposeManagementStatistics">true</attribute>

  </mbean>
</server>
```

The `CacheJmxWrapper` is not the cache itself (i.e. you can't store stuff in it). Rather, as it's name implies, it's a wrapper around an `org.jboss.cache.Cache` that handles integration with JMX. `CacheJmxWrapper` exposes the `org.jboss.cache.Cache` via its `CacheJmxWrapperMBean` MBean interfaces `Cache` attribute; services that need the cache can obtain a reference to it via that attribute.

11.2.3. Deployment Via a -jboss-beans.xml File

Much like it can deploy MBean services described with a -service.xml, JBoss AS 5 can also deploy services that consist of Plain Old Java Objects (POJOs) if the POJOs are described using the JBoss Microcontainer schema in a -jboss-beans.xml file. You create such a file and deploy it, either directly in the deploy dir, or packaged in an ear or sar. Following is an example:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- First we create a Configuration object for the cache -->
  <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

    <!-- Externally injected services -->
    <property name="runtimeConfig">
      <bean name="ExampleCacheRuntimeConfig"
            class="org.jboss.cache.config.RuntimeConfig">
        <property name="transactionManager">
          <inject bean="jboss:service=TransactionManager"
                  property="TransactionManager"/>
        </property>
        <property name="muxChannelFactory"><inject bean="JChannelFactory"/></property>
      </bean>
    </property>

    <property name="multiplexerStack">udp</property>
    <property name="clusterName">Example-EntityCache</property>
    <property name="isolationLevel">REPEATABLE_READ</property>
    <property name="cacheMode">REPL_SYNC</property>
    <property name="initialStateRetrievalTimeout">15000</property>
    <property name="syncReplTimeout">20000</property>
    <property name="lockAcquisitionTimeout">15000</property>
    <property name="exposeManagementStatistics">true</property>

  </bean>

  <!-- Factory to build the Cache. -->
  <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
    <constructor factoryClass="org.jboss.cache.DefaultCacheFactory" />
  </bean>

  <!-- The cache itself -->
```

```
  <bean name="ExampleCache" class="org.jboss.cache.Cache">
    <constructor factoryMethod="createCache">
      <factory bean="DefaultCacheFactory"/>
      <parameter class="org.jboss.cache.config.Configuration">
        <inject bean="ExampleCacheConfig"/>
      </parameter>
      <parameter class="boolean">false</false>
    </constructor>
  </bean>

  <bean name="ExampleService" class="org.foo.ExampleService">
    <property name="cache"><inject bean="ExampleCache"/></property>
  </bean>

</deployment>
```

The bulk of the above is the creation of a JBoss Cache `Configuration` object; this is the same as what we saw in the configuration of the CacheManager service (see *Section 11.1.1, "Editing the CacheManager Configuration"*). In this case we're not using the CacheManager service as a cache factory, so instead we create our own factory bean and then use it to create the cache (the "ExampleCache" bean). The "ExampleCache" is then injected into a (fictitious) service that needs it.

An interesting thing to note in the above example is the use of the `RuntimeConfig` object. External resources like a `TransactionManager` and a JGroups `ChannelFactory` that are visible to the microcontainer are dependency injected into the `RuntimeConfig`. The assumption here is that in some other deployment descriptor in the AS, the referenced beans have already been described.

Using the configuration above, the "ExampleCache" cache will not be visible in JMX. Here's an alternate approach that results in the cache being bound into JMX:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- First we create a Configuration object for the cache -->
  <bean name="ExampleCacheConfig"
        class="org.jboss.cache.config.Configuration">

     .... same as above

  </bean>

  <bean name="ExampleCacheJmxWrapper" class="org.jboss.cache.jmx.CacheJmxWrapper">
```

```
   <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(
        name="foo:service=ExampleCacheJmxWrapper",
        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
        registerDirectly=true)
   </annotation>

   <property name="configuration"><inject bean="ExampleCacheConfig"/></property>

  </bean>

  <bean name="ExampleService" class="org.foo.ExampleService">
     <property name="cache"><inject bean="ExampleCacheJmxWrapper" property="cache"/></
property>
  </bean>

</deployment>
```

Here the "ExampleCacheJmxWrapper" bean handles the task of creating the cache from the configuration. `CacheJmxWrapper` is a JBoss Cache class that provides an MBean interface for a cache. Adding an <annotation> element binds the JBoss Microcontainer `@JMX` annotation to the bean; that in turn results in JBoss AS registering the bean in JMX as part of the deployment process.

The actual underlying `org.jboss.cache.Cache` instance is available from the `CacheJmxWrapper` via its `cache` property; the example shows how this can be used to inject the cache into the "ExampleService".