JBoss Application Server 5

2

Clustering Guide

Authors

Brian Stansberry

Galder Zamarreno

ISBN:

Publication date: Nov 2008

This book is the Jboss Application Server 5 Clustering Guide.

JBoss Application Server 5: Clustering Guide: Authors

by Brian Stansberry, Galder Zamarreno, and Samson Kittoli

Clustering

High Availability Enterprise Services via JBoss Clusters

1. Introduction

Clustering allows us 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 simply 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. The simplest way to start a JBoss server cluster is to start several JBoss instances on the same local network, using the `run -c all` command for each instance. Those server instances, all started in the `all` configuration, detect each other and automatically form a cluster.

In the first section of this chapter, we discuss basic concepts behind JBoss's clustering services. It is important that you understand these concepts before reading the rest of the chapter. Clustering configurations for specific types of applications are covered after this section.

2. 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. In summary, a JBoss cluster is a set of AS server instances each of which is running an identically configured and named JGroups Channel.

On the same AS instance, different services can create their own Channel. In a default 5.0.x AS, four different services create channels – the web session replication service, the EJB3 SFSB replication service, the EJB3 entity caching service, and a core general purpose clustering service known as HAPartition. In order to differentiate these channels, each must have a unique name, and its configuration must match its peers yet differ from the other channels.

So, if you go to two AS 5.0.x instances and execute `run -c all`, the channels 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 4 channels, and hence 4 two node clusters.

On the same network, even for the same service, we may have different clusters. *Figure 1.1, "Clusters and server nodes"* shows an example network of JBoss server instances divided into three clusters, with the third cluster 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.

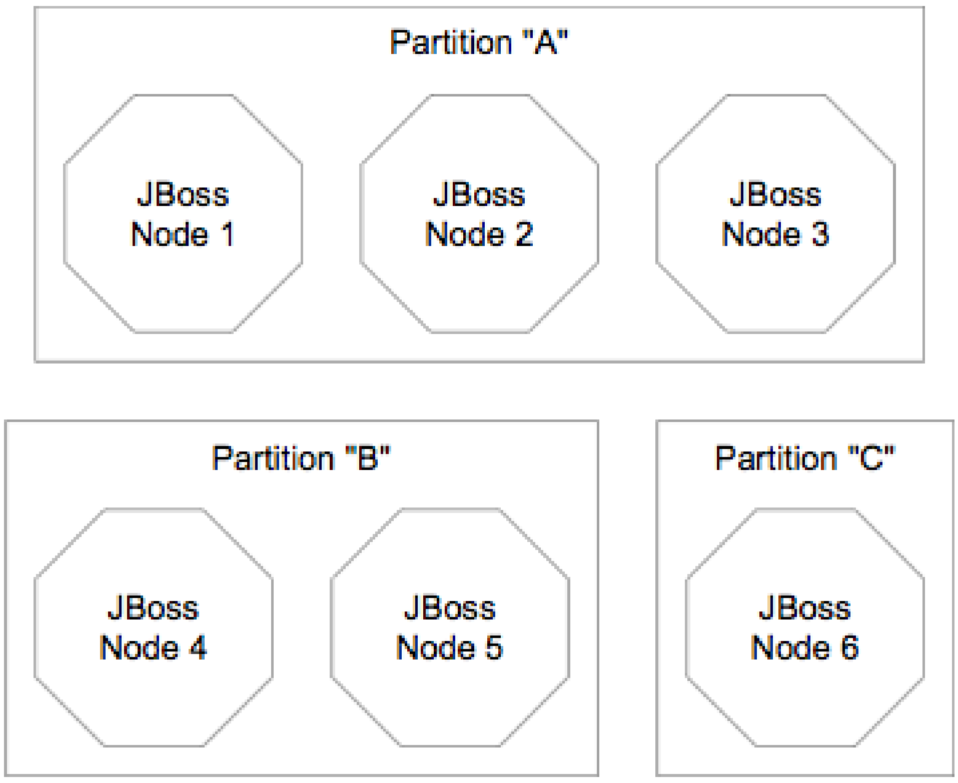

Figure 1.1. Clusters and server nodes

The section on "JGroups Configuration" and on "Isolating JGroups Channels" covers in detail how to configure Channels such that desired peers find each other and unwanted peers do not. As mentioned above, by default JBoss AS uses four separate JGroups Channels. These can be divided into two broad categories: the Channel used by the general purpose HAPartition service, and three Channels created by JBoss Cache for special purpose caching and cluster wide state replication.

3. HAPartition

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 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.

The following example shows the `HAPartition` MBean definition packaged with the standard JBoss AS distribution. So, if you simply start JBoss servers with their default clustering settings on a local network, you would get a default cluster named `DefaultPartition` that includes all server instances as its nodes.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
    name="jboss:service=DefaultPartition">

    <! -- Name of the partition being built -->
    <attribute name="PartitionName">
        ${jboss.partition.name:DefaultPartition}
    </attribute>

    <! -- The address used to determine the node name -->
    <attribute name="NodeAddress">${jboss.bind.address}</attribute>

    <! -- Determine if deadlock detection is enabled -->
    <attribute name="DeadlockDetection">False</attribute>

    <! -- Max time (in ms) to wait for state transfer to complete.
        Increase for large states -->
    <attribute name="StateTransferTimeout">30000</attribute>

    <! -- The JGroups protocol configuration -->
    <attribute name="PartitionConfig">
        ... ...
    </attribute>
</mbean>
```

Here, we omitted the detailed JGroups protocol configuration for this channel. JGroups handles the underlying peer-to-peer communication between nodes, and its configuration is discussed in *Section 1, "JGroups Configuration"*. The following list shows the available configuration attributes in the `HAPartition` MBean.

- **PartitionName** is an optional attribute to specify 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 an optional attribute used to help generate a unique name for this node.
- **DeadlockDetection** is an optional boolean attribute that tells JGroups to run message deadlock detection algorithms with every request. Its default value is `false`.
- **StateTransferTimeout** is an optional attribute to specify the timeout for state replication across the cluster (in milliseconds). State replication refers to the process of obtaining initial application state from other already-running cluster members at service startup. Its default value is `30000`.
- **PartitionConfig** is an element to specify JGroup configuration options for this cluster (see *Section 1, "JGroups Configuration"*).

In order for nodes to form a cluster, they must have the exact same `PartitionName` and the `ParitionConfig` elements. Changes in either element on some but not all nodes would cause the cluster to split.

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=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.

4. JBoss Cache channels

JBoss Cache is a fully featured distributed cache framework that can be used in any application server environment or standalone. JBoss AS integrates JBoss Cache to provide cache services for HTTP sessions, EJB 3.0 session beans, and EJB 3.0 entity beans. Each of these cache services is defined in a separate Mbean, and each cache creates its own JGroups Channel. We will cover those MBeans when we discuss specific services in the next several sections.

4.1. Service Architectures

The clustering topography defined by the `HAPartition` MBean on each node is of great importance to system administrators. But for most application developers, you are probably more concerned about 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.

4.1.1. 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 stub (or proxy) object. The stub object is generated by the server and it implements the business interface of the service. The client then makes local method calls against the stub object. The stub automatically routes the call across the network and where it is invoked against service objects managed in the server. In a clustering environment, the server-generated stub object includes an interceptor that understands how to route calls to multiple nodes in the cluster. The stub 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.

The stub interceptors maintain up-to-date knowledge about the cluster. For instance, they know 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 stub interceptor with the latest changes in the cluster. For instance, if a node drops out of the cluster, each of client stub interceptor is updated with the new configuration the next time it connects to any active node in the cluster. All the manipulations done by the service stub are transparent to the client application. The client-side interceptor clustering architecture is illustrated in *Figure 1.2, “The client-side interceptor (proxy) architecture for clustering”*.

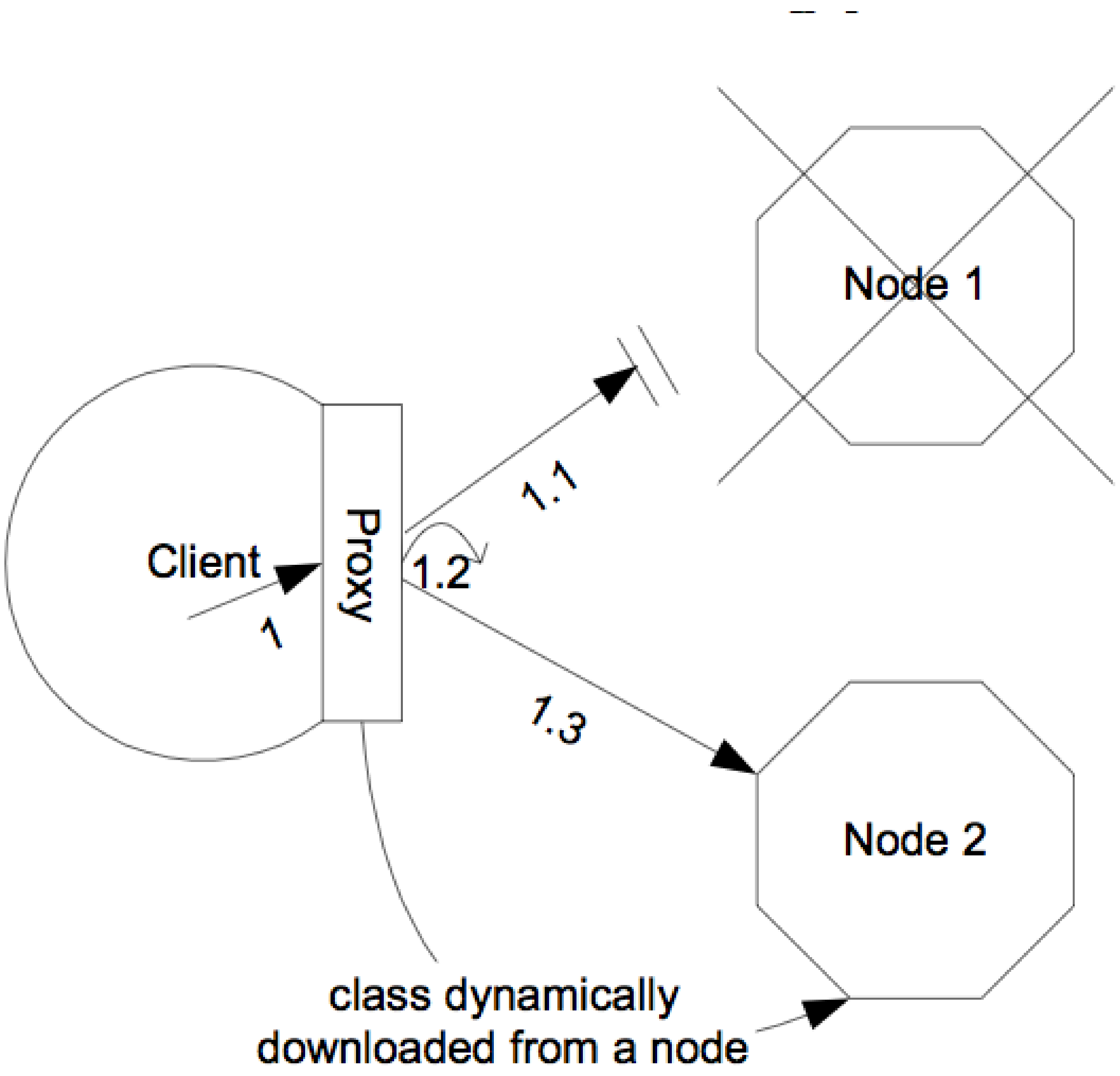

Figure 1.2. The client-side interceptor (proxy) architecture for clustering

Note

Section 1, “Stateless Session Bean in EJB 2.x” describes how to enable the client proxy to handle the entire cluster restart.

4.1.2. Load balancer

Other JBoss services, in particular the HTTP-based 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 according to certain communication protocols (e.g., 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 about 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 1.3, “The external load balancer architecture for clustering”*.

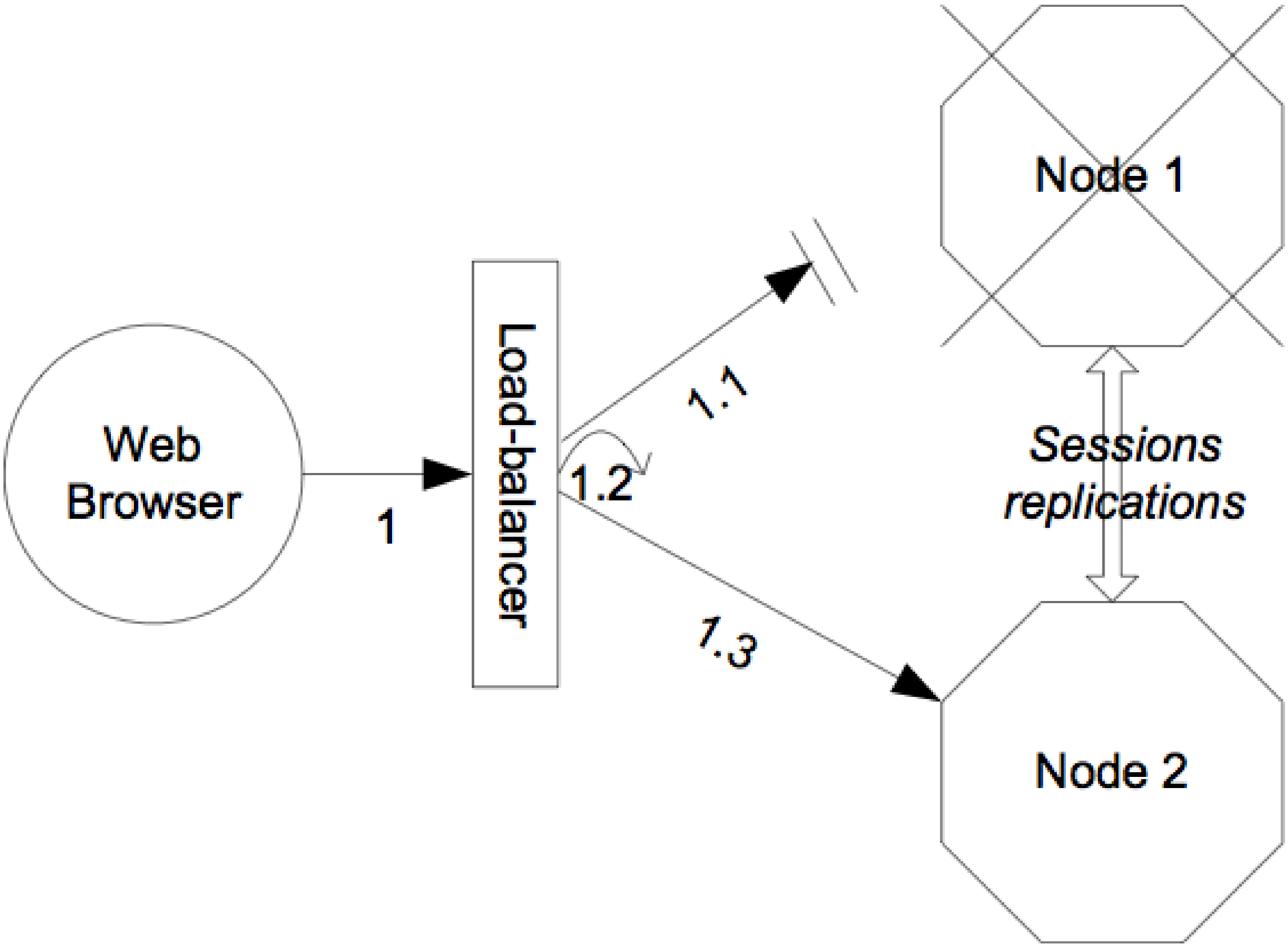

Figure 1.3. The external load balancer architecture for clustering

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.

4.2. Load-Balancing Policies

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

4.2.1. Client-side interceptor architecture

In JBoss 5.0.0, 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.

- Round-Robin (`org.jboss.ha.framework.interfaces.RoundRobin`): 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.
- Random-Robin (`org.jboss.ha.framework.interfaces.RandomRobin`): for each call the target node is randomly selected from the list.
- First Available (`org.jboss.ha.framework.interfaces.FirstAvailable`): 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 stub elects its own target node independently of the other stubs, so if a particular client downloads two stubs for the same target service (e.g., an EJB), each stub 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.
- First Available Identical All Proxies (`org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies`): has the same behaviour as the "First Available" policy but the elected target node is shared by all stubs in the same client-side VM that are associated with the same target service. So if a particular client downloads two stubs for the same target service (e.g. an EJB), each stub will use the same target.

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

4.2.2. 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 affinitiy” (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.

4.3. Farming Deployment

The easiest way to deploy an application into the cluster is to use the farming service. That is to hot-deploy the application archive file (e.g., the EAR, WAR or SAR file) in the `all/farm/` directory of any of the cluster members and the application will be automatically duplicated across all nodes in the same cluster. If 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 one of the running cluster server node's `farm/` folder, the application will be undeployed locally and then removed from all other cluster server nodes farm folder (triggers undeployment.) You should manually delete the application from the farm folder of any server node not currently connected to the cluster.

Note

Currently, due to an implementation weakness, the farm deployment service only works for 1) archives located in the farm/ directory of the first node to join the cluster or 2) hot-deployed archives. If you first put a new application in the farm/ directory and then start the server to have it join an already running cluster, the application will not be pushed across the cluster or deployed. This is because the farm service does not know whether the application really represents a new deployment or represents an old deployment that was removed from the rest of the cluster while the newly starting node was off-line. We are working to resolve this issue.

Note

You can only put zipped archive files, not exploded directories, in the farm directory. If exploded directories are placed in farm the directory contents will be replicated around the cluster piecemeal, and it is very likely that remote nodes will begin trying to deploy things before all the pieces have arrived, leading to deployment failure.

Note

Farmed deployment is not atomic. A problem deploying, undeploying or redeploying an application on one node in the cluster will not prevent the deployment, undeployment or redeployment being done on the other nodes. There is no rollback capability. Deployment is also not staggered; it is quite likely, for example, that a redeployment will happen on all nodes in the cluster simultaneously, briefly leaving no nodes in the cluster providing service.

Farming is enabled by default in the `all` configuration in JBoss AS distributions, so you will not have to set it up yourself. The `farm-service.xml` configuration file is located in the deploy/deploy.last directory. If you want to enable farming in a custom configuration, simply copy the farm-service.xml file and copy it to the JBoss deploy directory `$JBOSS_HOME/server/your_own_config/deploy/deploy.last`. Make sure that your custom configuration has clustering enabled.

After deploying farm-service.xml you are ready to rumble. The required FarmMemberService MBean attributes for configuring a farm are listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<server>

    <mbean code="org.jboss.ha.framework.server.FarmMemberService"


 name="jboss:service=FarmMember,partition=DefaultPartition">
        ...

 <depends optional-attribute-name="ClusterPartition"
 proxy-type="attribute">
  jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <attribute name="ScanPeriod">5000</attribute>
  <attribute name="URLs">farm/</attribute>
 ...
 </mbean>
</server>
```

- **ClusterPartition** is a required attribute to inject the HAPartition service that the farm service uses for intra-cluster communication.
- **URLs** points to the directory where deployer watches for files to be deployed. This MBean will create this directory is if does not already exist. If a full URL is not provided, it is assumed that the value is a filesytem path relative to the configuration directory (e.g. `$JBOSS_HOME/server/all/`).
- **ScanPeriod** specifies the interval at which the folder must be scanned for changes.. Its default value is `5000`.

The farming service is an extension of the `URLDeploymentScanner`, which scans for hot deployments in the `deploy/` directory. So, you can use all the attributes defined in the `URLDeploymentScanner` MBean in the `FarmMemberService` MBean. In fact, the `URLs` and `ScanPeriod` attributes listed above are inherited from the `URLDeploymentScanner` MBean.

4.4. Distributed state replication services

In a clustered server environment, distributed state management is a key service the cluster must provide. For instance, in a stateful session bean application, the session state must be synchronized among all bean instances across all nodes, so that the client application reaches the same session state no matter which node serves the request. In an entity bean application, the bean object sometimes needs to be cached across the cluster to reduce the database load. Currently, the state replication and distributed cache services in JBoss AS are provided via three ways: the `HASessionState` Mbean, the `DistributedState` MBean and the JBoss Cache framework.

- The `HASessionState` MBean is a legacy service that provides session replication and distributed cache services for EJB 2.x stateful session beans. The MBean is defined in the `all/deploy/cluster-service.xml` file. We will show its configuration options in the EJB 2.x stateful session bean section later.
- The `DistributedState` Mbean is a legacy service built on the HAPartition service. It is supported for backwards compatibility reasons, but new applications should not use it; they should use the much more sophisticated JBoss Cache instead.
- As mentioned above JBoss Cache is used to provide cache services for HTTP sessions, EJB 3.0 session beans and EJB 3.0 entity beans. It is the primary distributed state management tool in JBoss AS, and is an excellent choice for any custom caching requirements your applications may have. We will cover JBoss Cache in more detail when we discuss specific services in the next several sections..

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. 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 the JNDI so that the client can lookup 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.

1. How it works

The JBoss client-side HA-JNDI naming Context is based on the 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 the "Client Configuration" section. 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, he 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 that node is able to find objects bound into the local JNDI context tree. An application can bind its objects to either 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 (this is actually basic JNDI). 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 Context 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 services 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 (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

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.

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 that a lookup will hit a HA-JNDI server that does not own this binding is higher 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.

So, an EJB home lookup through HA-JNDI, will always be delegated to the local JNDI instance. If different beans (even of the same type, but participating in different clusters) use the same JNDI name, it means that each JNDI server will have a different "target" bound (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!

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 though of using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

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 to lookup a HA-JNDI server that does not own this binding is higher and 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.

2. Client configuration

2.1. For clients running inside the application server

If you want to access HA-JNDI from inside the application server, you must explicitly get an InitialContext by passing in JNDI properties. 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");
```

```
p.put(Context.PROVIDER_URL, "localhost:1100"); // HA-JNDI port.
return new InitialContext(p);
```

The Context.PROVIDER_URL property points to the HA-JNDI service configured in the HANamingService MBean (see the section called “JBoss configuration”).

However, this does not work in all cases, especially when running a multihomed cluster (several JBoss instances on one machine bound to different IPs). A safer method is not to specify the Context.PROVIDER_URL (which does not work in all scenarios) but the partition name 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");
p.put("jnp.partitionName", "DefaultPartition"); // partition name.
return new InitialContext(p);
```

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.

Note

Previously, HANamingServiceMBean.bindAddress served two functions: From `trunk/cluster/src/etc/hajndi-service.xml`:

```
<!-- Bind address of bootstrap and HA-JNDI RMI
 endpoints -->
  <attribute
 name="BindAddress">${jboss.bind.address}</attribute>
```

The bootstrap and HA-JNDI RMI endpoints are now defined separately:

```

  <!-- Bind address of bootstrap endpoint -->
  <attribute
 name="BindAddress">${jboss.bind.address}</attribute>
  <!-- Bind address of the HA-JNDI RMI endpoint -->
  <attribute
 name="RmiBindAddress">${jboss.bind.address}</
attribute>
```

They each default to the same value. Users may want to override the RMI bind address if deployed on a multi-homed machine, and want to use an specific network interface for HA-JNDI RMI calls. This ability already exists in the standard NamingService.

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 mapped to a URL that makes use of HA-JNDI.

```
<resource-ref>
 <res-ref-name>jms/ConnectionFactory</res-ref-name>
 <jndi-name>jnp://localhost:1100/ConnectionFactory</jndi-name>
</resource-ref>

<resource-ref>
 <res-ref-name>jms/Queue</res-ref-name>
 <jndi-name>jnp://localhost: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.

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.

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]

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 2.3, “JBoss configuration”* for how to configure the servers and ports).

```
java.naming.provier.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.

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 the section called “JBoss configuration” on 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 port1102.

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 select the first HA-JNDI server that responds, irregardless of the cluster partition name.

- `jnp.discoveryTimeout`: Determines how much time 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.
- `jnp.discoveryPort`: Determines which multicast group 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) 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.

2.3. JBoss configuration

The `cluster-service.xml` file in the `all/deploy` directory includes the following MBean to enable HA-JNDI services.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
name="jboss:service=HAJNDI">
<depends optional-attribute-name="ClusterPartition"
proxy-
type="attribute">jboss:service=${jboss.partition.name:DefaultPartition}</
depends>

<mbean>
```

You can see that this MBean depends on the `DefaultPartition` MBean defined above it (discussed earlier in this chapter). In other configurations, you can put that element in the `jboss-service.xml` file or any other JBoss configuration files in the `/deploy` directory to enable HA-JNDI services. The available attributes for this MBean are listed below.

- **Cluster Partition** is a required attribute to inject the HAPartition service that HA-JNDI uses for intra-cluster communication.
- **BindAddress** is an optional attribute to specify the address to which the HA-JNDI server will bind waiting for JNP clients. Only useful for multi-homed computers. The default value is the value of the jboss.bind.address system property, or the host's default addresss 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** is an optional attribute to specify the port to which the HA-JNDI server will bind waiting for JNP clients. The default value is `1100`.
- **Backlog** is an optional attribute to specify the backlog value used for the TCP server socket waiting for JNP clients. The default value is `50`.
- **RmiPort** determines which port the server should use to communicate with the downloaded stub. This attribute is optional. The default value is 1101. If no value is set, the server automatically assigns a RMI port.
- `DiscoveryDisabled` is a boolean flag that disables configuration of the auto discovery multicast listener.
- **AutoDiscoveryAddress** is an optional attribute to specify 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** is an optional attribute to specify the multicast group to listen to for JNDI automatic discovery.. 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 the earlier section on “Load-Balancing Policies” for details.
- **LookupPool** specifies the thread pool service used to control the bootstrap and auto discovery lookups.

The full default configuration of the `HANamingService` MBean is as follows.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
name="jboss:service=HAJNDI">
 <!-- We now inject the partition into the HAJNDI service instead
 of requiring that the partition name be passed -->
 <depends optional-attribute-name="ClusterPartition"

proxy-
```

```
  type="attribute">jboss:service=${jboss.partition.name:DefaultPartition}</
  depends>
    <!-- Bind address of bootstrap and HA-JNDI RMI endpoints -->
    <attribute name="BindAddress">${jboss.bind.address}</attribute>
    <!-- Port on which the HA-JNDI stub is made available -->
    <attribute name="Port">1100</attribute>
    <!-- RmiPort to be used by the HA-JNDI service once bound. 0 =>
   auto. -->
    <attribute name="RmiPort">1101</attribute>
    <!-- Accept backlog of the bootstrap socket -->
    <attribute name="Backlog">50</attribute>
    <!-- The thread pool service used to control the bootstrap and
   auto discovery lookups -->
   <depends optional-attribute-name="LookupPool"
    proxy-type="attribute">jboss.system:service=ThreadPool</depends>
    <!-- A flag to disable the auto discovery via multicast -->
   <attribute name="DiscoveryDisabled">false</attribute>
   <!-- Set the auto-discovery bootstrap multicast bind address. If
   not
    specified and a BindAddress is specified, the BindAddress will be
   used. -->
    <attribute
   name="AutoDiscoveryBindAddress">${jboss.bind.address}</attribute>
    <!-- Multicast Address and group port used for auto-discovery -->

    <attribute
   name="AutoDiscoveryAddress">${jboss.partition.udpGroup:230.0.0.4}</
  attribute>
    <attribute name="AutoDiscoveryGroup">1102</attribute>
    <!-- The TTL (time-to-live) for autodiscovery IP multicast
   packets -->
    <attribute name="AutoDiscoveryTTL">16</attribute>
    <!-- The load balancing policy for HA-JNDI -->
    <attribute
   name="LoadBalancePolicy">org.jboss.ha.framework.interfaces.RoundRobin</
  attribute>

    <!-- Client socket factory to be used for client-server
    RMI invocations during JNDI queries
    <attribute name="ClientSocketFactory">custom</attribute>
    -->
    <!-- Server socket factory to be used for client-server
    RMI invocations during JNDI queries
    <attribute name="ServerSocketFactory">custom</attribute>
     -->
     </mbean>
```

It is possible to start several HA-JNDI services that use different clusters. This can be used, for example, if a node is part of many clusters. In this case, make sure that you set a different port or IP address for eachservices. For instance, if you wanted to hook up HA-JNDI to the example cluster you set up and change the binding port, the Mbean descriptor would look as follows.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
      name="jboss:service=HAJNDI">

      <depends optional-attribute-name="ClusterPartition"

 proxy-type="attribute">jboss:service=MySpecialPartition</depends>

 <attribute name="Port">56789</attribute>
</mbean>
```

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 exactly the same as the client for the non-clustered version of the session bean, except for a minor change to the `java.naming.provier.url` system property to enable HA-JNDI lookup (see previous section). 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 2.x and EJB 3.0 server applications respectively.

1. Stateless Session Bean in EJB 2.x

Clustering stateless session beans is most probably the easiest case: as no state is involved, calls can be load-balanced on any participating node (i.e. any node that has this specific bean deployed) of the cluster. To make a bean clustered, you need to modify its `jboss.xml` descriptor to contain a `<clustered>` tag.

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>nextgen.StatelessSession</ejb-name>
            <jndi-name>nextgen.StatelessSession</jndi-name>
            <clustered>True</clustered>
            <cluster-config>
                <partition-name>DefaultPartition</partition-name>

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

                </home-load-balance-policy>
                <bean-load-balance-policy>
                    org.jboss.ha.framework.interfaces.RoundRobin
                </bean-load-balance-policy>
            </cluster-config>
        </session>
    </enterprise-beans>
</jboss>
```

Note

The `<clustered>True</clustered>` element is really just an alias for the `<configuration-name>Clustered Stateless`

SessionBean</configuration-name> element in the conf/standard-jboss.xml file.

In the bean configuration, only the <clustered> element is mandatory. It indicates that the bean needs to support clustering features. The <cluster-config> element is optional and the default values of its attributes are indicated in the sample configuration above. Below is a description of the attributes in the <cluster-config> element..

- **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. You can also implement your own load-balance policy class or use the class `FirstAvailable` that persists to use the first node available that it meets until it fails.
- **bean-load-balance-policy** Indicates the class to be used by the bean stub to balance calls made on the nodes of the cluster. Comments made for the `home-load-balance-policy` attribute also apply.

2. 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 `HASessionState` MBean 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 `HASessionState` MBean configuration.

2.1. 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-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>
                  <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 `HASessionState` service 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.

2.2. 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.

2.3. The HASessionState service configuration

The `HASessionState` service MBean is defined in the `all/deploy/cluster-service.xml` file.

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

    <depends>jboss:service=Naming</depends>
   <!-- We now inject the partition into the HAJNDI service instead

 of requiring that the partition name be passed -->
 <depends optional-attribute-name="ClusterPartition"
  proxy-type="attribute">
  jboss:service=${jboss.partition.name:DefaultPartition}
  </depends>
  <!-- JNDI name under which the service is bound -->
  <attribute name="JndiName">/HASessionState/Default</attribute>
  <!-- Max delay before cleaning unreclaimed state.
Defaults to 30*60*1000 => 30 minutes -->
<attribute name="BeanCleaningDelay">0</attribute>
</mbean>
```

The configuration attributes in the `HASessionState` MBean are listed below.

- **ClusterPartition** 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 `HASessionState` service is bound. The default value is `/HAPartition/Default`.
- **BeanCleaningDelay** is an optional attribute to specify the number of miliseconds after which the `HASessionState` service 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 `HASessionState` service needs to do this cleanup sometimes. The default value is `30*60*1000` milliseconds (i.e., 30 minutes).

2.4. 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>
```

2.5. 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.

2.6. 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.

3. Stateless Session Bean in EJB 3.0

To cluster a stateless session bean in EJB 3.0, all you need to do is to annotate the bean class withe the `@Clustered` annotation. You can pass in the load balance policy and cluster partition as parameters to the annotation. The default load balance policy is `org.jboss.ha.framework.interfaces.RandomRobin` and the default cluster is `DefaultPartition`. Below is the definition of the `@Cluster` annotation.

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

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
   }
}
```

The `@Clustered` annotation can also be omitted and the clustering configuration applied 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>
```

4. Stateful Session Beans in EJB 3.0

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the `@Cluster` annotation, just as we did with the EJB 3.0 stateless session bean earlier. The @org.jboss.ejb3.annotation.cache.tree.CacheConfig annotation can also be applied to the bean to specify caching behavior. Below is the definition of the @CacheConfig annotation:

```
public @interface CacheConfig
{
String name() default "jboss.cache:service=EJB3SFSBClusteredCache";
int maxSize() default 10000;
long idleTimeoutSeconds() default 300;
boolean replicationIsPassivation() default true;
long removalTimeoutSeconds() default 0;
}
```

- `name` specifies the object name of the JBoss Cache Mbean that should be used for caching the bean (see below for more on this Mbean).

- `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 also be omitted and the clustering configuration applied in jboss.xml; see the example above.

As with EJB 2.0 clustered SFSBs, JBoss provides a mechanism whereby a bean implementation can expose a method the container can invoke to check whether the bean's state is not dirty after a request and doesn't need to be replicated. With EJB3, the mechanism is a little more formal; instead of just exposing a method with a known signature, an EJB3 SFSB must implement the org.jboss.ejb3.cache.Optimized interface:

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

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The related MBean service is defined in the `ejb3-clustered-sfsbcache-service.xml` file in the `deploy` directory. The contents of the file are as follows.

```
<server>
 <mbean code="org.jboss..cache.TreeCache"
 name="jboss.cache:service=EJB3SFSBClusteredCache">

  <attribute name="ClusterName">
   ${jboss.partition.name:DefaultPartition}-SFSBCache
   </attribute>
   <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
   <attribute name="CacheMode">REPL_ASYNC</attribute>

   <!-- We want to activate/inactivate regions as beans are
 deployed -->
    <attribute name="UseRegionBasedMarshalling">true</attribute>
   <!-- Must match the value of "useRegionBasedMarshalling" -->
   <attribute name="InactiveOnStartup">true</attribute>

   <attribute name="ClusterConfig">
   ... ...
   </attribute>

   <!-- The max amount of time (in milliseconds) we wait until the
   initial state (ie. the contents of the cache) are retrieved from

   existing members.  -->
   <attribute name="InitialStateRetrievalTimeout">17500</attribute>

   <!--  Number of milliseconds to wait until all responses for a
    synchronous call have been received.
    -->
   <attribute name="SyncReplTimeout">17500</attribute>

   <!--  Max number of milliseconds to wait for a lock acquisition
 -->
   <attribute name="LockAcquisitionTimeout">15000</attribute>

    <!--  Name of the eviction policy class. -->
   <attribute name="EvictionPolicyClass">
    org.jboss.cache.eviction.LRUPolicy
   </attribute>

   <!--  Specific eviction policy configurations. This is LRU -->
   <attribute name="EvictionPolicyConfig">
    <config>
    <attribute name="wakeUpIntervalSeconds">5</attribute>
     <name>statefulClustered</name>
    <!-- So default region would never timeout -->
    <region name="/_default_">
    <attribute name="maxNodes">0</attribute>
     <attribute name="timeToIdleSeconds">0</attribute>
    </region>
   </config>
  </attribute>

 <!-- Store passivated sessions to the file system -->
  <attribute name="CacheLoaderConfiguration">
 <config>

  <passivation>true</passivation>
 <shared>false</shared>

   <cacheloader>
   <class>org.jboss.cache.loader.FileCacheLoader</class>
  <!-- Passivate to the server data dir -->
   <properties>
   location=${jboss.server.data.dir}${/}sfsb
  </properties>
  <async>false</async>
  <fetchPersistentState>true</fetchPersistentState>
  <ignoreModifications>false</ignoreModifications>
  </cacheloader>

    </config>
    </attribute>
 </mbean>
</server>
```

The configuration attributes in this MBean are essentially the same as the attributes in the standard JBoss Cache `TreeCache` MBean discussed in *Chapter 7, JBossCache and JGroups Services*. Again, we omitted the JGroups configurations in the `ClusterConfig` attribute (see more in *Section 1, “JGroups Configuration”*). Two noteworthy items:

- The 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.
- 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 others data.

Clustered Entity EJBs

In a JBoss AS cluster, the 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.

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.

1. 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 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.

2. Entity Bean in EJB 3.0

In EJB 3.0, the 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.

2.1. 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 Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying second-level cache implementation. 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 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 updated in the cache.

- If you remove an entity bean instance from the database via the entity manager the entity will 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.

The JBoss Cache service for EJB 3.0 entity beans is configured in a `TreeCache` MBean in the `deploy/ejb3-entity-cache-service.xml` file. The name of the cache MBean service is `jboss.cache:service=EJB3EntityTreeCache`. Below are the contents of the `ejb3-entity-cache-service.xml` file in the standard JBoss distribution. Again, we omitted the JGroups configuration element `ClusterConfig`.

```
<server>
 <mbean code="org.jboss.cache.TreeCache"
name="jboss.cache:service=EJB3EntityTreeCache">

 <depends>jboss:service=Naming</depends>
 <depends>jboss:service=TransactionManager</depends>

 <!-- Name of cluster. Needs to be the same on all nodes in the
clusters,
       in order to find each other -->
  <attribute name="ClusterName">
   ${jboss.partition.name:DefaultPartition}-EntityCache
  </attribute>

  <!-- Configure the TransactionManager -->
 <attribute name="TransactionManagerLookupClass">
   org.jboss.cache.JBossTransactionManagerLookup
 </attribute>

 <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
 <attribute name="CacheMode">REPL_SYNC</attribute>

 <!-- Must be true if any entity deployment uses a scoped
classloader -->
 <attribute name="UseRegionBasedMarshalling">true</attribute>
 <!-- Must match the value of "useRegionBasedMarshalling" -->
 <attribute name="InactiveOnStartup">true</attribute>

 <attribute name="ClusterConfig">
  ... ...
 </attribute>

 <attribute name="InitialStateRetrievalTimeout">17500</attribute>
 <attribute name="SyncReplTimeout">17500</attribute>
 <attribute name="LockAcquisitionTimeout">15000</attribute>
```

```
    <attribute name="EvictionPolicyClass">
    org.jboss.cache.eviction.LRUPolicy
    </attribute>

    <!--  Specific eviction policy configurations. This is LRU -->
     <attribute name="EvictionPolicyConfig">
     <config>
     <attribute name="wakeUpIntervalSeconds">5</attribute>
     <!--  Cache wide default -->
      <region name="/_default_">
      <attribute name="maxNodes">5000</attribute>
      <attribute name="timeToLiveSeconds">1000</attribute>
      </region>
     </config>
    </attribute>
    </mbean>
  </server>
```

This is a replicated cache, so, if running within a cluster, and the cache is updated, changes to the entries in one node will be replicated to the corresponding entries in the other nodes in the cluster.

JBoss Cache allows you to specify timeouts to cached entities. Entities not accessed within a certain amount of time are dropped from the cache in order to save memory. The above configuration sets up a default configuration region that says that at most the cache will hold 5000 nodes, after which nodes will start being evicted from memory, least-recently used nodes last. Also, if any node has not been accessed within the last 1000 seconds, it will be evicted from memory. In general, a node in the cache represents a cached item (entity, collection, or query result set), although there are also a few other node that are used for internal purposes. If the above values of 5000 maxNodes and 1000 idle seconds are invalid for your application(s), you can change the cache-wide defaults. You can also add separate eviction regions for each of your entities; more on this below.

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.

2.2. Configure the entity beans for cache

You define your entity bean classes the normal way. Future versions of JBoss EJB 3.0 will support annotating entities and their relationship collections as cached, but for now you have to configure the underlying hibernate engine directly. Take a look at the `persistence.xml` file, which configures the caching options for hibernate via its optional `property` elements. The following element in `persistence.xml` defines that caching should be enabled:

```
<!-- Clustered cache with TreeCache -->
<property name="cache.provider_class">
    org.jboss.ejb3.entity.TreeCacheProviderHook
</property>
```

The following property element defines the object name of the cache to be used, i.e., the name of the TreeCache MBean shown above.

```
<property name="treecache.mbean.object_name">
    jboss.cache:service=EJB3EntityTreeCache
</property>
```

Finally, you should give a “region_prefix” to this configuration. This ensures that all cached items associated with this persistence.xml are properly grouped together in JBoss Cache. The jboss.cache:service=EJB3EntityTreeCache cache is a shared resource, potentially used by multiple persistence units. The items cached in that shared cache need to be properly grouped to allow the cache to properly manage classloading. <property name="hibernate.cache.region_prefix" value="myprefix"/>

If you do not provide a region prefix, JBoss will automatically provide one for you, building it up from the name of the EAR (if any) and the name of the JAR that includes the persistence.xml. For example, a persistence.xml packaged in foo.ear, bar.jar would be given “foo_ear,bar_jar” as its region prefix. This is not a particularly friendly region prefix if you need to use it to set up specialized eviction regions (see below), so specifying your own region prefix is recommended.

Next we need to configure what entities be cached. 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 `ejb3-entity-cache-service.xml` configuration file. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache could evict oldest objects (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 `/myprefix/com/mycompany/entities/Account`.

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
   name="jboss.cache:service=EJB3EntityTreeCache">
    ... ...
   <attribute name="EvictionPolicyConfig">
    <config>
     <attribute name="wakeUpIntervalSeconds">5</attribute>
     <region name="/_default_">
      <attribute name="maxNodes">5000</attribute>
      <attribute name="timeToLiveSeconds">1000</attribute>
     </region>
    <!-- Separate eviction rules for Account entities -->
     <region name="/myprefix/com/mycompany/entities/Account">
      <attribute name="maxNodes">10000</attribute>
      <attribute name="timeToLiveSeconds">5000</attribute>
     </region>
    ... ...
   </config>
  </attribute>
 </mbean>
</server>
```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached in the `/_default` region 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:

```
<server>
 <mbean code="org.jboss.cache.TreeCache"
```

```
         name="jboss.cache:service=EJB3EntityTreeCache">
    ... ...
   <attribute name="EvictionPolicyConfig">
   <config>
    <attribute name="wakeUpIntervalSeconds">5</attribute>
    <region name="/_default_">
    <attribute name="maxNodes">5000</attribute>
    <attribute name="timeToLiveSeconds">1000</attribute>
     </region>
    <!-- Separate eviction rules for Account entities -->
     <region name="/myprefix/Account">
      <attribute name="maxNodes">10000</attribute>
      <attribute name="timeToLiveSeconds">5000</attribute>
     </region>
     ... ...
   </config>
   </attribute>
   </mbean>
  </server>
```

2.3. Query result caching

The EJB3 Query API also provides means for you to save in the second-level cache the results (i.e., collections of primary keys of entity beans, or collections of scalar values) of specified queries. 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:

```
<server>
   <mbean code="org.jboss.cache.TreeCache"
   name="jboss.cache:service=EJB3EntityTreeCache">
    ... ...
    <attribute name="EvictionPolicyConfig">
     <config>
     <attribute name="wakeUpIntervalSeconds">5</attribute>
      <region name="/_default_">
      <attribute name="maxNodes">5000</attribute>
      <attribute name="timeToLiveSeconds">1000</attribute>
      </region>
      <!-- Separate eviction rules for Account entities -->
      <region name="/myprefix/Account">
       <attribute name="maxNodes">10000</attribute>
       <attribute name="timeToLiveSeconds">5000</attribute>
      </region>
      <!-- Cache queries for 10 minutes -->
      <region
 name="/myprefix/org/hibernate/cache/StandardQueryCache">
       <attribute name="maxNodes">100</attribute>
       <attribute name="timeToLiveSeconds">600</attribute>
      </region>
      ... ...
     </config>
    </attribute>
   </mbean>
</server>
```

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:

```

<server>
 <mbean code="org.jboss.cache.TreeCache"
        name="jboss.cache:service=EJB3EntityTreeCache">
  ... ...
  <attribute name="EvictionPolicyConfig">
   <config>
    <attribute name="wakeUpIntervalSeconds">5</attribute>
    <region name="/_default_">
     <attribute name="maxNodes">5000</attribute>
     <attribute name="timeToLiveSeconds">1000</attribute>
    </region>
    <!-- Separate eviction rules for Account entities -->
    <region name="/myprefix/Account">
     <attribute name="maxNodes">10000</attribute>
     <attribute name="timeToLiveSeconds">5000</attribute>
    </region>
    <!-- Cache queries for 10 minutes -->
    <region name="/myprefix/Queries">
     <attribute name="maxNodes">100</attribute>
     <attribute name="timeToLiveSeconds">600</attribute>
    </region>
    ... ...
   </config>
  </attribute>
 </mbean>
</server>
```

HTTP Services

HTTP session replication is used to replicate the state associated with your web clients on other nodes of a cluster. Thus, in the event one of your node 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 below), 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. aThis 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: 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 query 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. In other situations, losing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

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.

2. 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 version 2.0.x. 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 developments are 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/`.

3. 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 server 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.

4. 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 (JBoss Tomcat) node1 and node2 listening on port 8009.

In the `works.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.

5. Configuring JBoss to work with mod_jk

Finally, we must configure the JBoss Tomcat 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/jboss-web.deployer/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.

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009" address="${jboss.bind.address}"
 protocol="AJP/1.3"
emptySessionPath="true" enableLookups="false" redirectPort="8443"
 />
```

Then, for each JBoss Tomcat instance in the cluster, we need to tell it that mod_jk is in use, so it can properly manage the `jvmRoute` appended to its session cookies so that mod_jk can properly route incoming requests. Edit the `JBOSS_HOME/server/all/deploy/jbossweb-tomcat50.sar/META-INF/jboss-service.xml` file (replace `/all` with your own server name). Locate the `<attribute>` element with a name of `UseJK`, and set its value to `true`:

```
<attribute name="UseJK">true</attribute>
```

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 Tomcat, please refer to the JBoss wiki page at `http://wiki.jboss.org/wiki/Wiki.jsp?page=UsingMod_jk1.2WithJBoss`.

6. 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 4, “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, the client can hit any server node and obtain the same session state.

The `jboss.cache:service=TomcatClusteringCache` MBean makes use of JBoss Cache to provide HTTP session replication services to the JBoss Tomcat cluster. This MBean is defined in the `deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml file`.

Note

Before AS 4.2.0, the location of the HTTP session cache configuration file was `deploy/tc5-cluster.sar/META-INF/jboss-service.xml`. Prior to AS 4.0.4 CR2, the file was named `deploy/tc5-cluster-service.xml`.

Below is a typical `deploy/jbossweb-cluster.sar/META-INF/jboss-service.xml` file. The configuration attributes in the `TomcatClusteringCache` MBean are very similar to those in the JBoss AS cache configuration.

```
<mbean code="org.jboss.cache.aop.TreeCacheAop"
    name="jboss.cache:service=TomcatClusteringCache">

    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>
    <depends>jboss.aop:service=AspectDeployer</depends>

    <attribute name="TransactionManagerLookupClass">
        org.jboss.cache.BatchModeTransactionManagerLookup
    </attribute>

    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>

    <attribute name="CacheMode">REPL_ASYNC</attribute>

    <attribute name="ClusterName">
      Tomcat-${jboss.partition.name:Cluster}
    </attribute>

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

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

    <attribute name="ClusterConfig">
        ... ...
    </attribute>


    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="SyncReplTimeout">20000</attribute>
</mbean>
```

Note that the value of the mbean element's code attribute is org.jboss.cache.aop.TreeCacheAop, which is different from the other JBoss Cache Mbeans used in JBoss AS. This is because FIELD granularity HTTP session replication (covered below) needs the added features of the `TreeCacheAop` (a.k.a. `PojoCache`) class.

The details of all the configuration options for a TreeCache MBean are covered in the JBoss Cache documentation. Below, we will just discuss several attributes that are most relevant to the HTTP cluster session replication.

- **TransactionManagerLookupClass** sets the transaction manager factory. The default value is `org.jboss.cache.BatchModeTransactionManagerLookup`. It tells the cache NOT to participate in JTA-specific transactions. Instead, the cache manages its own transactions. Please do not change this.

- **CacheMode** controls how the cache is replicated. The valid values are `REPL_SYNC` and `REPL_ASYNC`. With either setting the client request thread updates the local cache with the current sesssion contents and then sends a message to the caches on the other members of the cluster, telling them to make the same change. With REPL_ASYNC (the default) the request thread returns as soon as the update message has been put on the network. With REPL_SYNC, the request thread blocks until it gets a reply message from all cluster members, informing it that the update was successfully applied. Using synchronous replication makes sure changes are applied aroundthe cluster before the web request completes. However, synchronous replication is much slower.

- **ClusterName** specifies the name of the cluster that the cache works within. The default cluster name is the the word "Tomcat-" appended by the current JBoss partition name. All the nodes must use the same cluster name.

- The **UseMarshalling** and **InactiveOnStartup** attributes must have the same value. They must be `true` if `FIELD` level session replication is needed (see later). Otherwise, they are default to `false`.

- **ClusterConfig** configures the underlying JGroups stack. Please refer to *Section 1, "JGroups Configuration"* for more information.

- **LockAcquisitionTimeout** sets the maximum number of milliseconds to wait for a lock acquisition when trying to lock a cache node. The default value is 15000.

- **SyncReplTimeout** sets the maximum number of milliseconds to wait for a response from all nodes in the cluster when a synchronous replication message is sent out. The default value is 20000; should be a few seconds longer than LockAcquisitionTimeout.

7. Enabling session replication in your application

To enable clustering of your web application you must tag it 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. Here is an example:

```
<jboss-web>
    <replication-config>

 <replication-trigger>SET_AND_NON_PRIMITIVE_GET</replication-
trigger>
        <replication-granularity>SESSION</replication-granularity>

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

The `replication-trigger` element determines what triggers a session replication (i.e. when is a session is considered `dirty` and in need of replication). It has 4 options:

- **SET**: With this policy, the session is considered dirty only when an attribute is set in the session (i.e., HttpSession.setAttribute() is invoked.) If your application always writes changed values back into the session, this option will be most optimal in terms of performance. The downside of SET is that if an object is retrieved from the session and modified without being written back into the session, the session manager will not know the attribute is dirty and the change to that object may not be replicated.

- **SET_AND_GET**: With this policy, any attribute that is get or set will be marked as dirty. If an object is retrieved from the session and modified without being written back into the session, the change to that object will be replicated. The downside of SET_AND_GET is that it can have significant performance implications, since even reading immutable objects from the session (e.g., strings, numbers) will mark the read attributes as needing to be replicated.

- **SET_AND_NON_PRIMITIVE_GET**: This policy is similar to the SET_AND_GET policy except that get operationsthat return attribute values with primitive types do not mark the attribute as dirty. Primitive system types (i.e., String, Integer, Long, etc.) are immutable, so there is no reason to mark an attribute with such a type as dirty just because it has been read. If a get operation returns a value of a non-primitive type, the session manager has no simple way to know whether the object is mutable, so it assumes it is an marks the attribute as dirty. This setting avoids the downside of SET while reducing the performance impact of SET_AND_GET. It is the default setting.

- **ACCESS**: This option causes the session to be marked as dirty whenever it is accessed. Since a the session is accessed during each HTTP request, it will be replicated with each request. The purpose of ACCESS is to ensure session last-access timestamps are kept in sync around the cluster.. Since with the other replication-trigger options the time stamp may not be updated in other clustering nodes because of no replication, the session in other nodes may expire before the active node if the HTTP request does not retrieve or modify any session attributes. When this option is set, the session timestamps will be synchronized throughout the cluster nodes. Note that use of this option can have a significant performance impact, so use it with caution. With the other replication-trigger options, if a session has gone 80% of its expiration interval without being replicated, as a safeguard its timestamp will be replicated no matter what. So, ACCESS is only useful in special circumstances where the above safeguard is considered inadequate.

The `replication-granularity` element controls the size of the replication units. The supported values are:

- **ATTRIBUTE**: Replication is only for the dirty attributes in the session plus some session data, like the last-accessed timestamp. For sessions that carry large amounts of data, this option can increase replication performance. However, attributes will be separately serialized, so if there are any shared references between objects stored in the attributes, those shared references may be broken on remote nodes. For example, say a Person object stored under key “husband” has a reference to an Address, while another Person object stored under key “wife” has a reference to that same Address object. When the “husband” and “wife” attributes are separately deserialized on the remote nodes, each Person object will now have a reference to its own Address object; the Address object will no longer be shared.
- **SESSION**: The entire session object is replicated if any attribute is dirty. The entire session is serialized in one unit, so shared object references are maintained on remote nodes. This is the default setting.
- **FIELD**: 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 `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. Default is `true`.

If your sessions are generally small, SESSION is the better policy. If your session is larger and some parts are infrequently accessed, ATTRIBUTE replication will be more effective. If your application has very big data objects in session attributes and only fields in those objects are frequently modified, the FIELD policy would be the best. In the next section, we will discuss exactly how the FIELD level replication works.

8. 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.aop.AopMarker
public class Address
{
...
}
```

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

```
@org.jboss.cache.aop.InstanceOfAopMarker
public class Person
{
...
}
then when you have a sub-class like
public class Student extends Person
{
...
}
```

There will be no need to annotate `Student`. It will be annotated automatically because it is a sub-class of `Person`. Jboss AS 4.2 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:

```
/*
 * My usual comments here first.
 * @@org.jboss.web.tomcat.tc5.session.AopMarker
 */
public class Address
{
...
}
```

The anologue for `@InstanceAopMarker` is:

```
/*
 *
 * @@org.jboss.web.tomcat.tc5.session.InstanceOfAopMarker
 */
public class Person
{
...
}
```

Once you have annotated your classes, you will need to perform a pre-processing step to bytecode enhance your classes for use by TreeCacheAop. 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 on 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.

Note

You can see a complete example on how to build, deploy, and validate a FIELD-level replicated web application from this page: *http://wiki.jboss.org/wiki/Wiki.jsp?page=Http_session_field_level_example*. The example bundles the pre- and post-compile tools so you do not need to download JBoss AOP separately.

When you deploy the web application into JBoss AS, make sure that the following configurations are correct:

- In the server's `deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml` file, the `inactiveOnStartup` and `useMarshalling` attributes must both be `true`.

- In the application's `jboss-web.xml` file, the `replication-granularity` attribute must be `FIELD`.

Finally, let's see an example on how to use FIELD-level replication on those data classes. Notice that 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.

```
// Do this only once. So this can be in init(), e.g.
if(firstTime)
{
  Person joe = new Person("Joe", 40);
  Person mary = new Person("Mary", 30);
  Address addr = new Address();
  addr.setZip(94086);

  joe.setAddress(addr);
  mary.setAddress(addr); // joe and mary share the same address!

  session.setAttribute("joe", joe); // that's it.
  session.setAttribute("mary", mary); // that's it.
}

Person mary = (Person)session.getAttribute("mary");
mary.getAddress().setZip(95123); // this will update and replicate
 the zip code.
```

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

9. Monitoring session replication

If you have deployed and accessed your application, go to the `jboss.cache:service=TomcatClusteringCache` MBean and invoke the `printDetails` operation. You should see output resembling the following.

```
/JSESSION

/localhost

/quote

/FB04767C454BAB3B2E462A27CB571330
VERSION: 6
```

```
FB04767C454BAB3B2E462A27CB571330:
 org.jboss.invocation.MarshalledValue@1f13a81c

/AxCI8Ovt5VQTfNyYy9Bomw**
VERSION: 4
AxCI8Ovt5VQTfNyYy9Bomw**:
 org.jboss.invocation.MarshalledValue@e076e4c8
```

This output shows two separate web sessions, in one application named *quote*, that are being shared via JBossCache. This example uses a `replication-granularity` of `session`. Had `ATTRIBUTE` level replication been used, there would be additional entries showing each replicated session attribute. In either case, the replicated values are stored in an opaque `MarshelledValue` container. There aren't currently any tools that allow you to inspect the contents of the replicated session values.
If you do not see any output, either the application was not correctly marked as `distributable` or you haven't accessed a part of application that places values in the HTTP session. The `org.jboss.cache` and `org.jboss.web` logging categories provide additional insight into session replication useful for debugging purposes.

10. Using Clustered Single Sign On

JBoss supports clustered single sign-on, allowing a user to authenticate to one web application on a JBoss server and to be recognized on all web applications, on that same machine or on another node in the cluster, that are deployed on the same virtual host. Authentication replication is handled by the same JBoss Cache Mbean that is used by the HTTP session replication service. Although session replication does not need to be explicitly enabled for the applications in question, the `jboss-web-cluster.sar` file needs to be deployed.

To enable single sign-on, you must add the `ClusteredSingleSignOn` valve to the appropriate `Host` elements of the tomcat `server.xml` file. The valve configuration is shown here:

```
<Valve
 className="org.jboss.web.tomcat.tc5.sso.ClusteredSingleSignOn" />
```

11. 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.

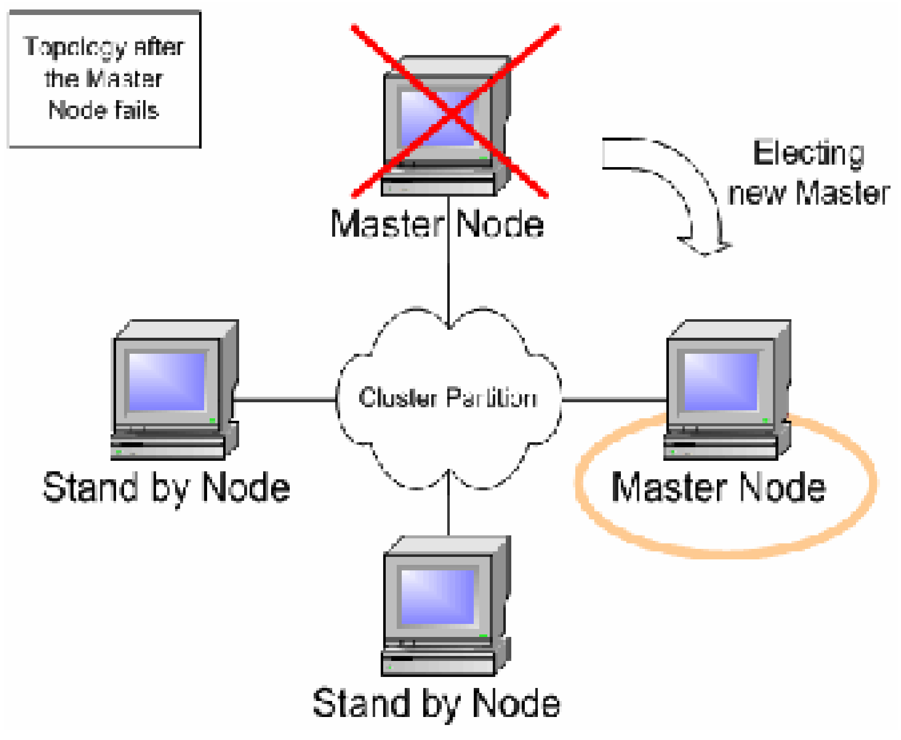

Figure 5.1. Topology after the Master Node fails

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.

11.1. 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 or farm, 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 `jboss.ha:service=HASingletonDeployer` MBean (which itself is deployed via the deploy/deploy-hasingleton-service.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 how complex the deployment of your service is and what sorts of startup activities it engages in, this could take a while, during which time the service is not being provided.

11.2. Mbean deployments using HASingletonController

If your service is an Mbean (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 an MBean service that we want to make an HA singleton. The only thing special about it is it needs to expose in its MBean interface a method that can be called when it should begin providing service, and another that can be called when it should stop providing service:

```

public class HASingletonExample
implements HASingletonExampleMBean {

private boolean isMasterNode = false;

public void startSingleton() {
isMasterNode = true;
}
.
public boolean isMasterNode() {
return isMasterNode;
 }
```

```
   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-service.xml`:

```

   <server>
    <!-- This MBean is an example of a clustered singleton -->
    <mbean code="org.jboss.ha.examples.HASingletonExample"
    name=“jboss:service=HASingletonExample"/>

    <!-- This HASingletonController manages the cluster Singleton -->

    <mbean code="org.jboss.ha.singleton.HASingletonController"
    name="jboss:service=ExampleHASingletonController">

     <!-- Inject a ref to the HAPartition -->
     <depends optional-attribute-name="ClusterPartition"
   proxy-type="attribute">
      jboss:service=${jboss.partition.name:DefaultPartition}
     </depends>
     <!-- Inject a ref to the service being controlled -->
     <depends optional-attribute-name="TargetName">
      jboss:service=HASingletonExample
     </depends>
     <!-- Methods to invoke when become master / stop being master
   -->
     <attribute name="TargetStartMethod">startSingleton</attribute>
     <attribute name="TargetStopMethod">stopSingleton</attribute>
    </mbean>
  </server>
```

Voila! A clustered singleton service.

The obvious downside to this approach is it only works for MBeans. Upsides are 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.

The jboss.ha:service=HASingletonDeployer service discussed above is itself an interesting example of using an HASingletonController. Here is its deployment descriptor (extracted from the `deploy/deploy-hasingleton-service.xml` file):

```
<mbean code="org.jboss.ha.singleton.HASingletonController"
name="jboss.ha:service=HASingletonDeployer">
 <depends optional-attribute-name="ClusterPartition"
 proxy-type="attribute">
  jboss:service=${jboss.partition.name:DefaultPartition}
 </depends>
 <depends optional-attributeame="TargetName">
  jboss.system:service=MainDeployer
 </depends>
 <attribute name="TargetStartMethod">deploy</attribute>
 <attribute name="TargetStartMethodArgument">
  ${jboss.server.home.url}/deploy-hasingleton
 </attribute>
 <attribute name="TargetStopMethod">undeploy</attribute>
 <attribute name="TargetStopMethodArgument">
  ${jboss.server.home.url}/deploy-hasingleton
 </attribute>
</mbean>
```

A few interesting things here. First the service being controlled is the `MainDeployer` service, which is the core deployment service in JBoss. That is, it's a service that wasn't written with an intent that it be controlled by an `HASingletonController`. But it still works! Second, the target start and stop methods are “deploy” and “undeploy”. No requirement that they have particular names, or even that they logically have “start” and “stop” functionality. Here the functionality of the invoked methods is more like “do” and “undo”. Finally, note the “`TargetStart(Stop)MethodArgument`” attributes. Your singleton service's start/stop methods can take an argument, in this case the location of the directory the `MainDeployer` should deploy/undeploy.

11.3. 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 mbean:

```
<depends>jboss.ha:service=HASingletonDeployer,type=Barrier</
depends>
```

The way it works is that a BarrierController is deployed along with the jboss.ha:service=HASingletonDeployer MBean 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 MBean 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 destroyed too.

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.

11.4. 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?

Prior to JBoss AS 4.2.0, the methodology for this was fixed and simple. For each member of the cluster, the HAPartition mbean 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 does this by checking if they are the first member of the view – if they are, they are the master; if not, they're not. Simple as that.

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.

JBoss Messaging Clustering Notes

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.

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.

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.

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.

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`.

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.

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.

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.

JBossCache and JGroups Services

JGroups and JBossCache provide the underlying communication, node replication and caching services, for JBoss AS clusters. Those services are configured as MBeans. There is a set of JBossCache and JGroups MBeans for each type of clustering applications (e.g., the Stateful Session EJBs, HTTP session replication etc.).

The JBoss AS ships with a reasonable set of default JGroups and JBossCache MBean configurations. Most applications just work out of the box with the default MBean configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements.

1. JGroups Configuration

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. It is built on top a stack of network communication protocols that provide transport, discovery, reliability and failure detection, and cluster membership management services. *Figure 7.1, “Protocol stack in JGroups”* shows the protocol stack in JGroups.

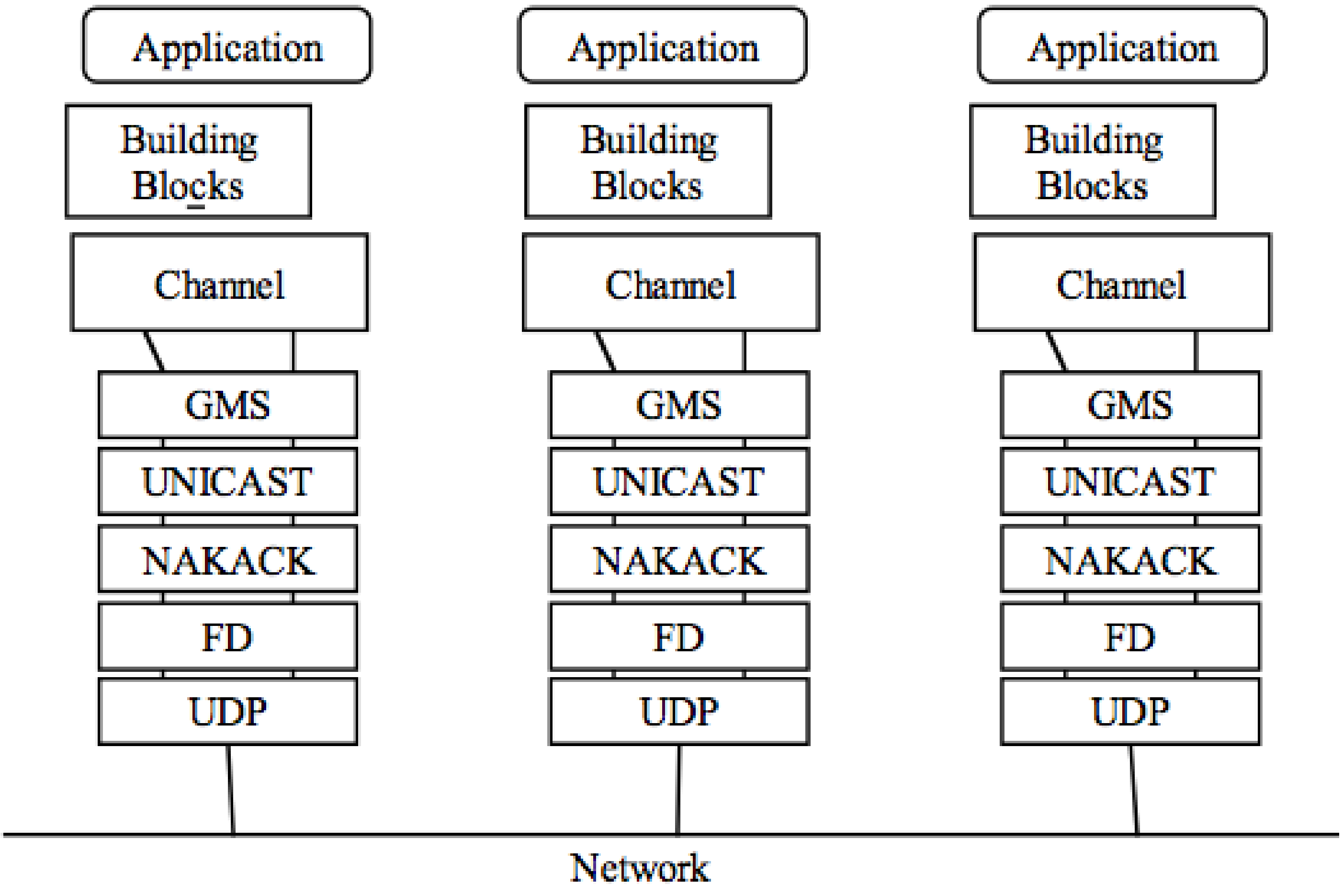

Figure 7.1. Protocol stack in JGroups

JGroups configurations often appear as a nested attribute in cluster related MBean services, such as the `PartitionConfig` attribute in the `ClusterPartition` MBean or the `ClusterConfig` attribute in the `TreeCache` MBean. You can configure the behavior and properties of each protocol in JGroups via those MBean attributes. Below is an example JGroups configuration in the `ClusterPartition` MBean.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
 name="jboss:service=${jboss.partition.name:DefaultPartition}">

  ... ...

  <attribute name="PartitionConfig">
   <Config>

    <UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
         mcast_port="${jboss.hapartition.mcast_port:45566}"
         tos="8"
```

```
           ucast_recv_buf_size="20000000"
           ucast_send_buf_size="640000"
           mcast_recv_buf_size="25000000"
           mcast_send_buf_size="640000"
           loopback="false"
           discard_incompatible_packets="true"
           enable_bundling="false"
           max_bundle_size="64000"
           max_bundle_timeout="30"
           use_incoming_packet_handler="true"
           use_outgoing_packet_handler="false"
           ip_ttl="${jgroups.udp.ip_ttl:2}"
           down_thread="false" up_thread="false"/>

      <PING timeout="2000"
            down_thread="false" up_thread="false"
   num_initial_members="3"/>

      <MERGE2 max_interval="100000"
       down_thread="false" up_thread="false" min_interval="20000"/>
      <FD_SOCK down_thread="false" up_thread="false"/>

      <FD timeout="10000" max_tries="5"
          down_thread="false" up_thread="false" shun="true"/>
      <VERIFY_SUSPECT timeout="1500" down_thread="false"
   up_thread="false"/>
      <pbcast.NAKACK max_xmit_size="60000"
       use_mcast_xmit="false" gc_lag="0"
       retransmit_timeout="300,600,1200,2400,4800"
       down_thread="false" up_thread="false"
       discard_delivered_msgs="true"/>
      <UNICAST timeout="300,600,1200,2400,3600"
        down_thread="false" up_thread="false"/>
      <pbcast.STABLE stability_delay="1000"
   desired_avg_gossip="50000"
       down_thread="false" up_thread="false"
       max_bytes="400000"/>
      <pbcast.GMS print_local_addr="true" join_timeout="3000"
           down_thread="false" up_thread="false"
           join_retry_timeout="2000" shun="true"
           view_bundling="true"/>
      <FRAG2 frag_size="60000" down_thread="false"
   up_thread="false"/>
      <pbcast.STATE_TRANSFER down_thread="false"
        up_thread="false" use_flush="false"/>
     </Config>
    </attribute>
  </mbean>
```

All the JGroups configuration data is contained in the <Config> element under the JGroups config MBean attribute. 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.

2. Common Configuration Properties

The following common properties are exposed by all of the JGroups protocols discussed below:

- `down_thread` whether the protocol should create an internal queue and a queue processing thread (aka the down_thread) for messages passed down from higher layers. The higher layer could be another protocol higher in the stack, or the application itself, if the protocol is the top one on the stack. If true (the default), when a message is passed down from a higher layer, the calling thread places the message in the protocol's queue, and then returns immediately. The protocol's down_thread is responsible for reading messages off the queue, doing whatever protocol-specific processing is required, and passing the message on to the next protocol in the stack.
- `up_thread` is conceptually similar to down_thread, but here the queue and thread are for messages received from lower layers in the protocol stack.

Generally speaking, `up_thread` and `down_thread` should be set to false.

3. Transport Protocols

The transport protocols send messages from one cluster node to another (unicast) or from cluster node to all other nodes in the cluster (mcast). JGroups supports UDP, TCP, and TUNNEL as transport protocols.

Note

The `UDP`, `TCP`, and `TUNNEL` elements are mutually exclusive. You can only have one transport protocol in each JGroups `Config` element

3.1. UDP configuration

UDP is the preferred protocol for JGroups. UDP uses multicast or 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 mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
     mcast_port="${jboss.hapartition.mcast_port:45566}"
     tos="8"
     ucast_recv_buf_size="20000000"
     ucast_send_buf_size="640000"
     mcast_recv_buf_size="25000000"
     mcast_send_buf_size="640000"
     loopback="false"
     discard_incompatible_packets="true"
     enable_bundling="false"
     max_bundle_size="64000"
     max_bundle_timeout="30"
     use_incoming_packet_handler="true"
     use_outgoing_packet_handler="false"
     ip_ttl="${jgroups.udp.ip_ttl:2}"
 down_thread="false" up_thread="false"/>
```

The available attributes in the above JGroups configuration are listed below.

- **ip_mcast** specifies whether or not to use IP multicasting. The default is `true`. If set to false, it 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 joining a group (i.e., the cluster). If omitted, the default is `228.8.8.8` .
- **mcast_port** specifies the multicast port number. If omitted, the default is `45566`.
- **bind_addr** specifies the interface on which to receive and send multicasts (uses the `-Djgroups.bind_address` system property, if present). If you have a multihomed machine, set the `bind_addr` attribute or system property to the appropriate NIC IP address. By default, system property setting takes priority over XML attribute unless -Djgroups.ignore.bind_addr system property is set.
- **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 to receive multicasts on. 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`".

- **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.
- **use_incoming_packet_handler** specifies whether to use a separate thread to process incoming messages. Sometimes receivers are overloaded (they have to handle de-serialization etc). Packet handler is a separate thread taking care of de-serialization, receiver thread(s) simply put packet in queue and return immediately. Setting this to true adds one more thread. The default is `true`.
- **use_outgoing_packet_handler** specifies whether to use a separate thread to process outgoing messages. The default is false.
- **enable_bundling** specifies whether to enable message bundling. If it is `true`, the node would queue outgoing messages until `max_bundle_size` bytes have accumulated, or `max_bundle_time` milliseconds have elapsed, whichever occurs first. Then bundle queued messages into a large message and send it. The messages are unbundled at the receiver. The default is `false`.
- **loopback** specifies whether to loop outgoing message back up the stack. In `unicast` mode, the messages are sent to self. In `mcast` mode, a copy of the mcast message is sent. The default is `false`
- **discard_incompatibe_packets** specifies whether to discard packets from different JGroups versions. 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 discarded if set to true, otherwise a warning will be logged. The default is `false`
- **mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_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.
- `tos` specifies traffic class for sending unicast and multicast datagrams.

Note

On Windows 2000 machines, because of the media sense feature being broken with multicast (even after disabling media sense), you need to set the UDP protocol's `loopback` attribute to `true`.

3.2. 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 start_port="7800"
    bind_addr="192.168.5.1"
    loopback="true"
    down_thread="false" up_thread="false"/>
```

Below are the attributes available in the `TCP` element.

- **bind_addr** specifies the binding address. It can also be set with the `-Djgroups.bind_address` command line option at server startup.
- **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 from `start_port`. If no available port is found (e.g., because of a firewall) 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.
- **loopback** specifies whether to loop outgoing message back up the stack. In `unicast` mode, the messages are sent to self. In `mcast` mode, a copy of the mcast message is sent. The default is false.
- **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.

3.3. TUNNEL configuration

The TUNNEL protocol uses an external router to send messages. The external router is known as a `GossipRouter`. 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 router_port="12001"
    router_host="192.168.5.1"
    down_thread="false" up_thread="false/>
```

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.
- **loopback** specifies whether to loop messages back up the stack. The default is `true`.

4. Discovery Protocols

The cluster needs to maintain a list of current member nodes at all times so that the load balancer and client interceptor know how to route their requests. Discovery protocols are used to discover active nodes in the cluster and detect the oldest member of the cluster, which is the coordinator. All initial nodes are discovered when the cluster starts up. When a new node joins the cluster later, it is only discovered after the group membership protocol (GMS, see *Section 7.1, “Group Membership”*) admits it into the group.

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 MBean `Config` element.

4.1. 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="2"
    down_thread="false" up_thread="false"/>
```

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

```
<PING gossip_host="localhost"
      gossip_port="1234"
       timeout="3000"
       num_initial_members="3"
       down_thread="false" up_thread="false"/>
```

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-seperated list of addresses (e.g., `host1[12345],host2[23456]`), which are pinged for discovery.

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.

4.2. 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"
     initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"
     num_initial_members="3"
  down_thread="false" up_thread="false"/>
```

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-seperated list of addresses (e.g., `host1[12345],host2[23456]`) for GossipRouters to register with.

4.3. TCPPING

The TCPPING protocol takes a set of known members and ping 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"
 initial_hosts="hosta[2300],hostb[3400],hostc[4500]"
 port_range="3"
 num_initial_members="3"
         down_thread="false" up_thread="false"/>
```

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-seperated list of addresses (e.g., `host1[12345],host2[23456]`) for pinging.
- **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. The configuration options allows for multiple nodes on the same host to be pinged.

4.4. MPING

MPING uses IP multicast to discover the initial membership. It can be used with all transports, but usually this is used in combination with TCP. TCP usually requires TCPPING, which has to list all group members explicitly, but MPING doesn't have this requirement. The typical use case for this is when we want TCP as transport, but multicasting for discovery so we don't have to define a static list of initial hosts in TCPPING or require external Gossip Router.

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

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.
- **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.

5. 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 view so that the load balancer and client interceptors know to avoid the dead node. The failure detection protocols are configured as sub-elements in the JGroups MBean `Config` element.

5.1. FD

FD is a failure detection protocol based on heartbeat messages. This protocol requires each node to periodically send are-you-alive messages to its neighbour. If the neighbour 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 after which, if the node is still considered dead, updates the cluster's view. Here is an example FD configuration.

```
<FD timeout="2000"
    max_tries="3"
    shun="true"
    down_thread="false" up_thread="false"/>
```

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. Once shunned, the node will be expelled from the cluster even if it comes back later. The shunned node would have to re-join the cluster through the discovery process. JGroups allows to configure itself such that shunning leads to automatic rejoins and state transfer, which is the default behaivour within JBoss Application Server.

Note

Regular traffic from a node counts as if it is a live. So, the are-you-alive messages are only sent when there is no regular traffic to the node for sometime.

5.2. 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 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_down_thread="false" up_thread="false"/>
```

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

- **bind_addr** specifies the interface to which the server socket should bind to. If -Djgroups.bind_address system property is defined, XML value will be ignore. This behaivour can be reversed setting -Djgroups.ignore.bind_addr=true system property.

5.3. 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"
 down_thread="false" up_thread="false"/>
```

The available attributes in the FD_SOCK element are listed below.

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

5.4. 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 real failures and therefore 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 down_thread="false" up_thread="false"/>
<FD timeout="10000" max_tries="5" shun="true"
down_thread="false" up_thread="false" />
```

This suspects a member when the socket to the neighbor has been closed abonormally (e.g. process crash, because the OS closes all sockets). However, f a host or switch crashes, then the sockets won't be closed, therefore, as a seond line of defense, FD will suspect the neighbor after 50 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 ca 50 seconds.

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

6. Reliable Delivery Protocols

Reliable delivery protocols within the JGroups stack ensure that data pockets are actually 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.

6.1. UNICAST

The UNICAST protocol is used for unicast messages. It uses ACK. It is configured as a sub-element under the JGroups Config element. If the JGroups stack is configured with 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="100,200,400,800"
down_thread="false" up_thread="false"/>
```

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.

6.2. NAKACK

The NAKACK protocol is used for multicast messages. It uses NAK. Under this protocol, each message is tagged with a sequence number. The receiver keeps track of the sequence numbers and deliver the messages in order. When a gap in the sequence number is detected, the receiver asks the sender to retransmit the missing message. 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"
   down_thread="false" up_thread="false"/>
```

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

- **retransmit_timeout** specifies the retransmission timeout (in milliseconds). It is the same as the `timeout` attribute in the UNICAST protocol.
- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just the node requesting it. This is useful when the sender drops the pocket -- so we do not need to retransmit for each node.
- **max_xmit_size** specifies maximum size for a bundled retransmission, if multiple packets are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivery messages on the receiver nodes. By default, we save all delivered messages. 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 garbage collection lags behind.

7. Other Configuration Options

In addition to the protocol stacks, you can also configure JGroups network services in the `Config` element.

7.1. Group Membership

The group membership service 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 the load balancer and client side interceptors, 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"
    down_thread="false" up_thread="false"
    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 maximum number of milliseconds to wait after a failed JOIN to re-submit it.
- **print_local_addr** specifies whether to dump the node's own address to the output when started.
- **shun** specifies whether a node should shun itself if it receives a cluster view that it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node as the cluster coordinator.
- **view_bundling** specifies whether multiple JOIN or LEAVE request arriving at the same time are bundled and handled together at the same time, only sending out 1 new view / bundle. This is is more efficient than handling each request separately.

7.2. Flow Control

The flow control service tries to adapt the sending data rate and the receiving data among nodes. If a sender node is too fast, it might overwhelm the receiver node and result in dropped packets that have to be retransmitted. In JGroups, the 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 sends 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 service is configured in the `FC` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<FC max_credits="1000000"
down_thread="false" up_thread="false"
    min_threshold="0.10"/>
```

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 threshold credit on the sender, below which the receiver should send in more credits.
- **min_threshold** specifies percentage value of the threshold. It overrides the `min_credits` attribute.

Note

Applications that use synchronous group RPC calls primarily do not require FC protocol in their JGroups protocol stack because synchronous communication, where the hread that makes the call blocks waiting for responses from all the members of the group, already slows overall rate of calls. Even though TCP provides flow control by itself, FC is still required in TCP based JGroups stacks because of group communication, where we essentially have to send group messages at the highest speed the slowest receiver can keep up with. TCP flow control only takes into account individual node communications and has not a notion of who's the slowest in the group, which is why FC is required.

7.2.1. 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 is fast. When A sends a group message, it establishes TCP connections A-A (conceptually), A-B, A-C and A-D (if they don't yet exist). 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. (BTW: this is also the reason why we need NAKACK, although 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 of S. Because all members buffer all messages they received, they need to purge stable messages (= messages seen by everyone) every now and then. This is done by the STABLE protocol, which can be configured to run the stability protocol round time based (e.g. every 50s) or size based (whenever 400K data has been received).

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, we need to send messages at a rate the slowest receiver (D) can handle.

7.2.2. 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 makes synchronous group RPC calls (typically using a JGroups RpcDispatcher.) 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.)

7.3. Fragmentation

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" down_thread="false" up_thread="false"/>
```

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.

Note

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

7.4. State Transfer

The state transfer service transfers the state from an existing node (i.e., the cluster coordinator) to a newly joining node. 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 down_thread="false" up_thread="false"/>
```

7.5. Distributed Garbage Collection

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"
    down_thread="false" up_thread="false"
       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 this service.
- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Value `0` disables this service.
- **stability_delay** specifies delay before we send STABILITY msg (give others 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.

7.6. Merging

When a network error occurs, the cluster might be partitioned into several different partitions. JGroups has a MERGE service that allows the coordinators in partitions to communicate with each other and form a single cluster back again. The flow control service is configured in the `MERGE2` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<MERGE2 max_interval="10000"
    min_interval="2000"
    down_thread="false" up_thread="false"/>
```

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

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

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

Note

The cluster states are not merged in a merger. This has to be done by the application. 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 would not merge all the nodes even though shunning is disabled. Alternatively use MPING, which is commonly used with TCP to provide multicast member discovery capabilities, instead of TCPPING to avoid having to specify all the nodes.

7.7. 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 `bind_addr` to the desired interfaces.

7.8. Isolating JGroups Channels

Within JBoss AS, there are a number of services that independently create JGroups channels -- 3 different JBoss Cache services (used for HttpSession replication, EJB3 SFSB replication and EJB3 entity replication) along with 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, multicast address, and multicast port, so isolating JGroups channels comes down to ensuring different channels use different values for the group name, multicast address and multicast port.

To isolate JGroups channels for different services on the same set of AS instances from each other, you MUST change the group name and the multicast port. In other words, each channel must have its own set of values.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed along with a JBoss Cache used for web session clustering. The HAPartition channels should not communicate with the JBoss Cache channels. They should use a different group name and multicast port. They can use the same multicast address, although they don't need to.

To isolate JGroups channels for the same service from other instances of the service on the network, you MUST change ALL three values. Each channel must have its own group name, multicast address, and multicast port.

For example, say we have a production cluster of 3 machines, each of which has an HAPartition deployed. On the same network there is also a QA cluster of 3 machines, which also has an HAPartition deployed. The HAPartition group name, multicast address, and multicast port for the production machines must be different from those used on the QA machines.

7.9. Changing the Group Name

The group name for a JGroups channel is configured via the service that starts the channel. Unfortunately, different services use different attribute names for configuring this. For HAPartition and related services configured in the deploy/cluster-service.xml file, this is configured via a PartitionName attribute. For JBoss Cache services, the name of the attribute is ClusterName.

Starting with JBoss AS 4.0.4, for the HAPartition and all the standard JBoss Cache services, we make it easy for you to create unique groups names simply by 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,

```
<attribute
 name="ClusterName">Tomcat-${jboss.partition.name:Cluster}</
attribute>
```

7.10. Changing the multicast address and port

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 configs in JBoss AS:

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

Unfortunately, setting the multicast ports is not so simple. As described above, by default there are four separate JGroups channels in the standard JBoss AS all configuration, and each should be given a unique port. There are no command line switches to set these, but the standard configuration files do use system properties to set them. So, they can be configured from the command line by using -D. For example,

```
/run.sh -u 230.1.2.3 -g QAPartition
-Djboss.hapartition.mcast_port=12345
-Djboss.webpartition.mcast_port=23456
-Djboss.ejb3entitypartition.mcast_port=34567
-Djboss.ejb3sfsbpartition.mcast_port=45678 -b 192.168.1.100 -c all
```

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.

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

It should be sufficient to just change the address, but there is a problem on several operating systems whereby packets addressed to a particular multicast port are delivered to all listeners on that port, regardless of the multicast address they are listening on. So the recommendation is to change both the address and the port.

7.11. JGroups Troubleshooting

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.

7.12. 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.