Edition

7

Date: 2004-05-21, 4:00:32 PM

SACHA LABOUREY, BILL BURKE

The JBoss Group

JBoss AS Clustering

SACHA LABOUREY, BILL BURKE, AND JBOSS

JBoss AS Clustering

© JBoss, Inc
3340 Peachtree Road; Suite 1225
Atlanta, GA 30326 USA
sales@jboss.com

Table of Content

PREFACE **13**

FORWARD 13
ABOUT THE AUTHORS 13
DEDICATION 14
ACKNOWLEDGMENTS 14

0. INTRODUCTION TO CLUSTERING **15**

WHAT THIS BOOK COVERS 15

1. INTRODUCTION **17**

DEFINITIONS 17
JBOSS CLUSTERING FEATURES 19

2. CLUSTERING IN JBOSS: OVERVIEW **21**

PARTITIONS 21
SUB-PARTITIONS 23
SMART PROXIES 25
CONSEQUENCES FOR NON-RMI CLIENT 29
AUTOMAGIC NODE DISCOVERY 30
NETWORK COMMUNICATION 30

3. SETTING UP CLUSTERING **33**

4. HA-JNDI **41**

JNDI AND HA-JNDI 41
HA-JNDI SET UP 42
HA-JNDI BINDING AND LOOKUP RULES 45
HA-JNDI DESIGN NOTE 48
HA-JNDI CLIENT AND AUTO-DISCOVERY 48
HA-JNDI JNP SPECIFIC PROPERTIES 50

5. CLUSTERING EJB **53**

STATELESS SESSION BEANS 54
STATEFUL SESSION BEANS 55

ENTITY BEANS 58
Entity Synchronization *59*
MESSAGE DRIVEN BEANS 60
LOAD-BALANCE POLICIES 60
JBoss 3.0.x *60*
JBoss ≥ 3.2 ..... *62*

6. HTTP SESSION CLUSTERING 65

INTRODUCTION 65
Do you really need HTTP Sessions replication? *67*
HTTP SESSION REPLICATION SETUP 67
Introduction *67*
Apache and mod_jk *68*
Activating and configuring Tomcat Session Replication Service *73*
Activate session replication in your Web Application *78*
Session Replication Tuning *78*

7. FARMING 81

8. CACHE INVALIDATION 83

OVERVIEW 83
FRAMEWORK ARCHITECTURE 85
WHAT THE FRAMEWORK IS NOT 87
EJB INTEGRATION 88
EJB CONTAINER CONFIGURATION 90
EJB Container Configuration *90*
Bean Configuration *93*
BRIDGES 95
JMS-based Bridge *95*
JBossCluster-based Bridge *97*
USE CASES 98
Single JVM RO/RW bean *98*
RO/RW cluster *100*

9. CLUSTERING ARCHITECTURE 103

OVERVIEW 103
JBOSS CLUSTERING FRAMEWORK 104
HAPartition *104*
Distributed Replicant Manager (DRM) *109*
Distributed State (DS) *111*
HA-RMI *113*

10. CLUSTERING YOUR OWN SERVICES 119

11. OTHER CLUSTERING SERVICES 121

SINGLETON SERVICE 121
SCHEDULER SERVICE 129
NOTIFICATION SERVICE 132

12. TROUBLE SHOOTING AND LIMITATIONS 143

FIRST, ARE YOU A WINDOWS USER? 143
TROUBLE SHOOTING 145

IF ALL ELSE FAILS… 146
LIMITATIONS....147

13. INDEX 149

Table of Listings

Listing 3-1. Clustering MBean definition ____ 33
Listing 3-2. JGroups protocol stack configuration ____ 34
Listing 4-1. HA-JNDI MBean definition ____ 42
Listing 4-2. Overriding HA-JNDI default values ____ 44
Listing 4-3. Setting JNDI properties in code to access HA-JNDI ____ 47
Listing 4-4. Sample HA-JNDI property string for multiple known servers ____ 49
Listing 5-1. Setting a stateless session bean as clustered ____ 54
Listing 5-2. Session State MBean definition ____ 55
Listing 5-3. Setting a stateful session bean as clustered ____ 57
Listing 5-4. Setting an entity bean as clustered ____ 58
Listing 5-5. Recreating a new remote proxy for each call ____ 61
Listing 5-6 Reusing a remote proxy for each call ____ 61
Listing 6-1. Including mod_jk's configuration file in Apache's main configuration file (conf/httpd.conf) ____ 69
Listing 6-2. mod_jk's configuration file (conf/mod-jk.conf) ____ 70
Listing 6-3. conf/workers.properties sample file for 2 workers Servlet container ____ 72
Listing 9-1. Example of clustered code ____ 107
Listing 10-12-1. Non-multicast JGroups config ____ 147

Table of Figures

Figure 1. Partitions 22
Figure 2. In-memory backup in a two-nodes cluster 23
Figure 3. In-memory backup inside sub-partitions 24
Figure 4. Client-managed fail-over 25
Figure 5. Dispatcher-managed fail-over 26
Figure 6. Client proxy-managed fail-over 28
Figure 7. HA-JNDI name resolution 42
Figure 8. HA-JNDI detailed lookup resolution process 46
Figure 9HTTP Session failover 66
Figure 10 Cache Invalidation Framework Architecture 85
Figure 11Cache Invalidation Framework Integration in EJB 89
Figure 12RO/RW Cluster Using Cache Invalidations 100
Figure 13. JBoss clustering building blocks 104
Figure 14. Replicas in a cluster 106
Figure 15. State transfer process 108
Figure 16. Standard smart-stub to RMI server 114
Figure 17. HARMIServer coupled with a pre-existing smart stub 115
Figure 18. Clustered Singleton Service 122
Figure 19. Controller MBean View. "MasterNode" is True on only one of the nodes 127
Figure 20. Sample singleton MBean View. "MasterNode" has the same value as "MasterNode" on the controller MBean 128
Figure 21. Clustered Notification Service 132

Preface

Forward

JBossClustering originally began in April 2001 as a small prototype built on top of JGroups and coded by Sacha Labourey. Bill Burke joined Sacha in August 2001 and together they redesigned JBossClustering from scratch to its current form in JBoss 3.x.

About the Authors

Sacha Labourey is one of the core developers of JBoss Clustering and the General Manager of JBoss Group (Europe). He frequently gives advanced JBoss training courses. He owns a master in computer science from the Swiss Federal Institute of Technology and was the founder of Cogito Informatique, a Swiss company specializing in the application server and middleware fields.

Bill Burke is one of the core developers of JBoss Clustering and Chief Architect at JBoss Group. He gives regular talks at advanced JBoss training courses and seminars on the subject. Bill has over 9 years experience implementing and using middleware in the industry. He was one of the primary developers of Iona Technology's, Orbix 2000 CORBA product and has also designed and implemented J2EE applications at Mercantec, Dow Jones, and Eigner Corporation.

JBoss, Inc. is a privately held services company based out of Atlanta, Georgia with offices around the world. Its purpose is to provide support and services for the JBoss applications server technology directly from the core developers and generate revenues rewarding its employees and, most importantly, JBoss developers. JBoss Inc. was founded by the developers of JBoss.

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

Dedication

All your clustering are belong to us.

Acknowledgments

We would like to thank all developers that participate in the JBoss AS clustering project, including Bela Ban, Thomas Peuss and Ivelin Ivanov.

Chapter 0

0. Introduction to Clustering

What this Book Covers

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

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

The primary focus of this book is the presentation of the standard JBoss components introduced above, from both the perspective of their configuration and architecture. As a user of a standard JBoss distribution you will be given an understanding of how to configure the standard components. In addition, the final chapter presents a detailed discussion of building and deploying an enterprise application to help you master the details of packaging and deploying applications with JBoss.

Chapter 1

1. Introduction

Quick introduction to Clustering terminology in JBoss

The terminology used in clustering is often confused and dependent of the environment in which it is used. This chapter settles the definitions we will use in this book.

Definitions

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

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

The **availability** of a service is the proportion of time for which a particular service is accessible with reasonable/predictable response times. The term **High availability** is generally used to denote a "high" proportion. Nevertheless, this proportion is context-dependent: HA for a critical system in a space ship is most probably based on higher figure than HA for a regional web site. The HA proportion thus define the maximal allowed downtime in a particular period. Table 1 presents

some maximal allowed downtimes per year depending on the HA proportion.

HA Proportion	Maximal allowed cumulated downtime per year
98 %	7.30 days
99 %	87.60 hours
99.5 %	43.80 hours
99.9 %	8.76 hours
99.95 %	4.38 hours
99.99 %	53.00 minutes
99.999 %	5.25 minutes
99.9999 %	31.00 seconds
99.99999 %	3.10 seconds

Table 1 Sample allowed downtime per HA proportions

It is clear that even if HA Proportion is strictly relative to its associated allowed downtime, cost is generally not: passing from 99 % to 99.99 % is generally much more expensive than from 98% to 99% even if the difference is bigger in absolute.

For example, the Telco industry generally requires a "5-9" (i.e. 99.999 %) HA level.

Fault tolerance implies High availability. Nevertheless, Highly available data is not necessarily strictly correct data, whereas a fault tolerant service always guarantees strictly correct behaviour despite a certain number and type of faults.

Consequently, some systems *only* require high availability (directory service consisting of static data for example) whereas others require fault tolerance (banking systems requiring transactional reliability for example.)

Load balancing is a means to obtain better performance by dispatching incoming requests to different servers. It does not make any assumption on the level of fault tolerance or availability of the system. Thus, a web site could use a farm of servers to render complex output based on basic information stored in a database. If the database is not a bottleneck, load-balancing requests between servers in the farm would largely improve performances. Nevertheless, the database represents a single point of failure. A database outage would cause the entire server farm to be useless. In this case, growing the server farm does not improve the system availability.

Some systems are able to offer fault tolerant behaviour (and, consequently, high availability) and load balancing for better **scalability**.

For more information about distributed systems concepts, see [1].

JBoss Clustering Features

JBoss currently supports the following clustering features.

- Automatic discovery. Nodes in a cluster find each other with no additional configuration.

---

1 Distributed Systems, concepts and design, Coulouris, Dollimore and Kindberg, Addison-Wesley 2001.

- Fail-over and load-balancing features for:
    - JNDI,
    - RMI (can be used to implement your own clustered services),
    - Entity Beans,
    - Stateful Session Beans with in memory state replication,
    - Stateless Session Beans
- HTTP Session replication with Tomcat 4.1 and 5.0
- Dynamic JNDI discovery. JNDI clients can automatically discover the JNDI InitialContext.
- Cluster-wide replicated JNDI tree.
- Farming. Distributed cluster-wide hot-deployment. Hot deploy on one box, it gets farmed to all nodes in the cluster.
- Pluggable RMI load-balance policies.

Chapter 2

2. Clustering in JBoss: Overview

Overview of JBoss clustering architecture choices

This section will introduce the main concepts that form the foundation of the clustering features in JBoss.

In particular, you will see how clusters are defined, how they are built and how nodes communicate together.

JBoss currently provides full clustering support for stateless session beans, stateful session beans, entity beans and JNDI. Replication of HTTP sessions for web applications is also available.

Partitions

As previously discussed, a cluster is a set of nodes. In JBoss, a node is a JBoss server instance. Thus, to build a cluster, several JBoss instances have to be grouped in what we call a **partition**.

The **partition** is the central concept for clustering in JBoss.

On a same network, we may have different partitions. In order to differentiate them, each partition must have an individual name.

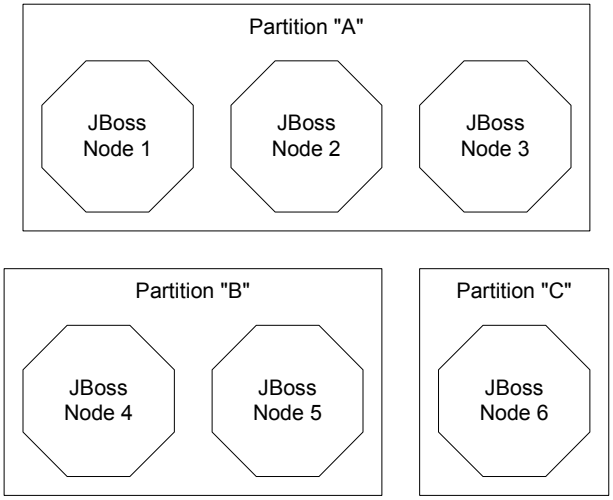


Figure 1. Partitions

In Figure 1, one of the clusters shows a limit case: it is composed of a single node. While this doesn't bring any particular interest (no fault tolerance or load balancing is possible), a new node could be added at any time to this partition which would then become much more interesting.

You will see later that it is possible for a JBoss instance to be part of multiple partitions at the same time. To simplify our discussion, we will currently consider that a JBoss server is always part of a single partition though.

If no name is assigned to a partition, it uses a default name. Consequently, simply starting new nodes without specifying a partition name (like in the default configuration file), will make all started nodes belong to the same default partition/cluster.

The current partition implementation in JBoss uses the JGroups framework2, but any communication framework offering a comparable semantic could be plugged in.

Sub-partitions

While a partition defines a set of nodes that work toward a same goal, some clustering features require to sub-partition the cluster to achieve a better scalability. Although JBoss does **not** currently support sub-partitioning, it will soon, so let's discuss it here.

For example, let's imagine that we want to replicate in memory the state of stateful session beans on different nodes to provide for fault-tolerant behaviour. In a two nodes cluster, this wouldn't be a challenge: each node would own a backup of all beans' state of the other node.

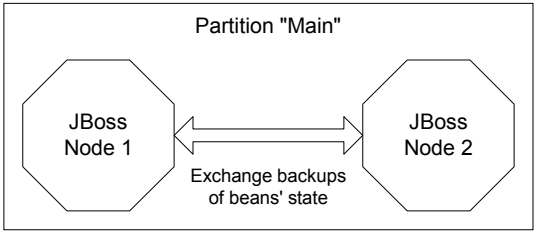


Figure 2. In-memory backup in a two-nodes cluster

But what if we had a 10-nodes cluster? With the previous scheme, it would mean that each node has to store a backup of the 9 other nodes. Firstly, this would not scale at all (each node would need to carry the whole state cluster load) and secondly we can wonder if any application

[2] http://www.JGroups.org/

would require this level of fault tolerance and still use Stateful Session Beans.

You can see from this example that it is probably much better to have some kind of sub-partitions inside a partition and have beans state exchanged only between nodes that are part of the same sub-partition.

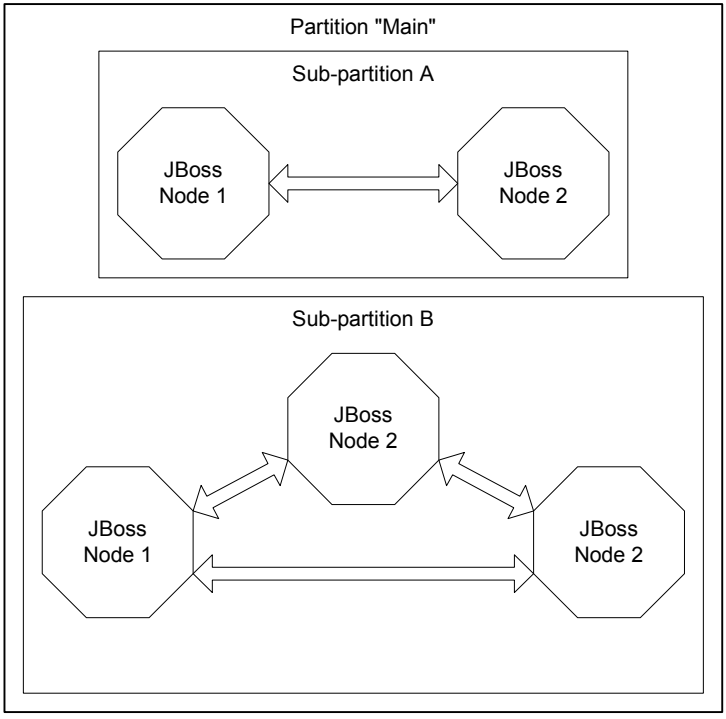


Figure 3. In-memory backup inside sub-partitions

In Figure 3, the "Main" partition has been subdivided in two sub-partitions. The cluster administrator wanted to have, ideally, sub-partitions containing two nodes. Nevertheless, in this particular case, if

the cluster had strictly followed this requirement, we would have had 3 sub-partitions with one owning only one node. Thus, this last sub-partition would not have any fault tolerant behaviour. Consequently, the cluster has decided to temporarily add this otherwise singleton node in an already full sub-partition. Thus, our fault tolerant objective is safe.

We sometimes name “brother nodes”, nodes belonging to the same sub-partition.

The future JBoss sub-partition implementation will allow the cluster administrator to determine the optimal size of a sub-partition. The sub-partition topology computation will be done dynamically by the cluster.

Smart proxies

Independently of the clustering solution you choose, the fail-over and load-balancing mechanism needs to take place somewhere.

When a client communicates with a node that suddenly fails, some solutions require that the client try to explicitly reconnect with a running node. While this kind of solution can provide fault tolerance, it is not transparent to the client.

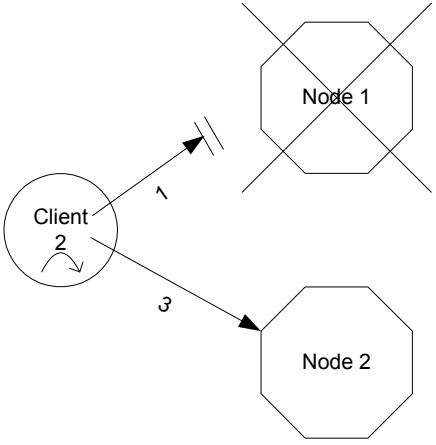


Figure 4. Client-managed fail-over

Consequently, we could imagine having some kind of transparent fail-over taking place on the server: a server receives the call (let's call it a "dispatcher"), dispatches it to the appropriate application server and if it fails (and if the semantic allows it), fail-over to another node. Nevertheless, we still have a problem with the dispatcher: what happens if it fails? This can looks like a chicken and egg problem…

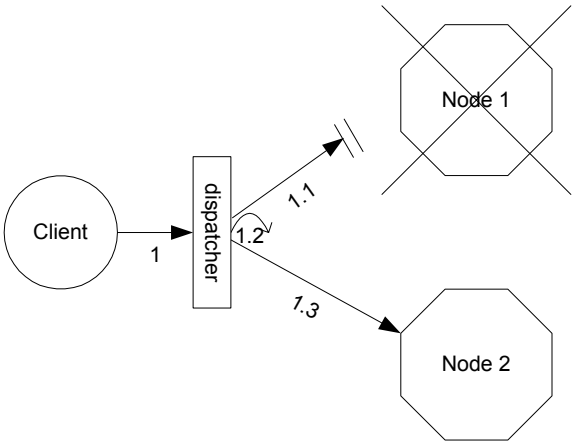


Figure 5. Dispatcher-managed fail-over

Luckily, RMI provides some nice features over many middleware protocols. For example, when a client gets a stub to a remote object, it is possible to send it a specialised serialised object. Furthermore, it is even possible to make the client download the class of the stub from any web server transparently. What happens is that when the client gets the remote reference, it obtains a serialised object annoted with a URL indicating from where the client RMI subsystem can download the class definition. Thus, this downloaded object:

- Acts as a stub for the remote reference,

- Can be generated programmatically at runtime and does not necessary needs to be part of the client libraries (JAR) at runtime (as it can be downloaded),

- Can incorporate some specific behaviour, in the stub code, that will transparently run on the client side (the client code doesn't even know about this code).

The first point is important because it represents the contract between the client and the target object: it expects to be able to make remote invocations on the target object through its reference.

The second point allows EJB applications to be deployed in JBoss without first generating, compiling and distributing client stubs.

The third point is a key point for the JBoss clustering implementation. It allows making the client download stub code that will contain the clustering logic i.e. fail-over and load-balancing logic. Thus, there is no risk to have the dispatcher fails as in the previous example. In this case, the dispatcher is incorporated in the client code. Thus, client and dispatcher life cycles are highly related (i.e. if the dispatcher dies, it most probably means that the client code is also dead and will not be much angry).

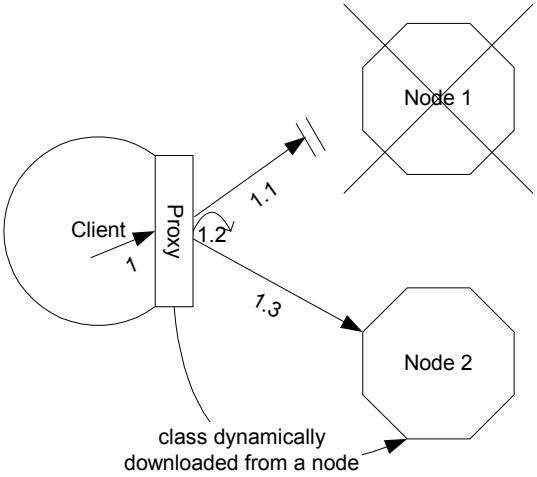


Figure 6. Client proxy-managed fail-over

In the current implementation, called HA-RMI, the proxy that is downloaded by the client code contains in particular the list of target nodes that it can access and a plugable load-balancing policy (round robin, first available, etc.) That means, for example, that a bean deployer can determine at deployment time how the proxy should behave (in fact, it could even change its decision while the container is already running).

Furthermore, if the cluster topology (i.e. the member of the partition to which the proxy is linked) changes, the next time the client performs an invocation, the JBoss server will piggy-back a new list of targets nodes with the invocation response. The proxy, before returning the response to the client code, will unpack the list of target nodes from the response, update its list of target nodes and return the real invocation result to the client code.

These features are available to any service or applications, not only JBoss specific components. For more architectural information and example on how to use HA-RMI, see section 0, “HA-RMI”.

Consequences for non-RMI client

Having the clustering logic embedded in RMI proxies has several consequences.

On the positive side, this solution gives a large number of possibilities regarding clustering policies and behaviour and it is transparent to the client code.

On the negative side, this implies that, at first sight, the solution is highly RMI-dependant. All non-RMI clients seem to loose JBoss clustering features.

In fact, the situation is not so dark. Looking at it closer, the model can very simply be degraded for non-RMI clients to the second model presented above: dispatcher-managed fail-over.

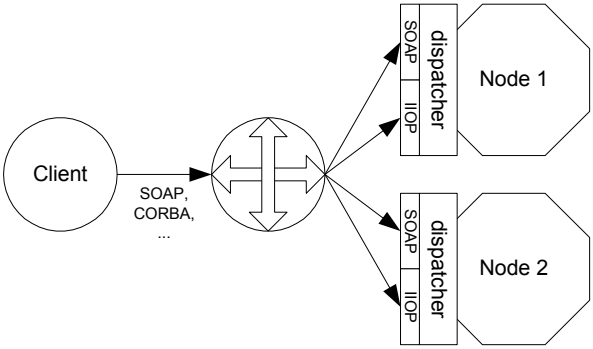
On the JBoss side of the (Java!) dispatcher, RMI is used to communicate with the cluster. Consequently, all features introduced until now apply.

On the client side of the dispatcher, any protocol end-point can be plugged in. The role of the dispatcher is then to translate this call from one protocol to the other3.

But there is still a problem: how to handle a dispatcher crash. In this case, since JBoss has no control over the client code, no software solution can be used. Consequently, a redundant hardware load-balancer has to be used as a front-end to the JBoss instances, to balance calls to the dispatchers.

---

[3] This information needs to be modified : with JBoss 3.0 detached invokers, this point may be completely dump because, thanks to our JMX bus, every JBoss node **is** a dispatcher. It already offers this separation.



Automagic Node Discovery

With JBoss clustering, there is no need to statically define a cluster topology: the only interesting information is the cluster name i.e. partition name (and not setting a partition name is valid because it will use a default name). Once the partition name has been set, any node can dynamically join or leave the partition.

Furthermore, as we have seen in the previous section, clients performing invocation on the cluster are automagically updated with a new view of the cluster members.

As we will see in the next section, node discovery and communication is highly configurable and can thus be applied to very different network topologies (LAN, WAN, etc.)

Network communication

Nodes in a cluster need to communicate together for numerous reasons :

- To detect new nodes or nodes that have failed/died,
- To exchange state information,

- To perform some action on a single, some or all distant nodes.

Consequently, several different activities need to take place at the network level.

The JBoss clustering framework has been abstracted in such a way that it should be possible to plug-in different communication frameworks.

Currently, JBoss uses the JGroups reliable communication framework.

This highly configurable framework uses the **Channel** as its basic building block. A Channel is a means by which a node can join a group of other nodes and exchange reliable messages in unicast or multicast. At runtime it is possible to build a new channel and specify, through a property string *à la* JDBC connection string (or through an XML snippet), the stack of protocols to use for this particular Channel: message fragmentation, state transfer, group membership.

Furthermore, at the lowest level, it is possible to decide whether to use TCP, UDP or UDP in multicast (or any other protocol) for network communications.

On top of the Channel building block, several helper building blocks are available, such as one-to-many reliable RPC calls.

For more information on the JGroups framework, check http://www.jgroups.org/.

Chapter 3

3. Setting up Clustering

Setting up partitions and configuring applications

As previously discussed, partitions are the basic building block of clustering in JBoss.

Each JBoss node participating in a cluster must at least have one partition. In JBoss terminology, a partition is equivalent to a cluster. Consequently, a node is able to participate in more than one cluster by starting more than one partition (each with a different name!) Nevertheless, in most cases, a node will be part of only one partition. Furthermore, each partition consumes resources (network bandwidth, CPU and threads), thus limiting the number of partition to a minimum is probably a good idea.

JBoss 3.0 comes with three different ready-to-use server configurations: *minimal*, *default* and *all*. Clustering is only enabled in the *all* configuration. A *cluster-service.xml* file in the */deploy* folder describes the configuration for the default cluster partition. This xml-snippet from *cluster-service.xml* loads the main partition MBean that initializes JGroups and other cluster components.

Listing 3-1. Clustering MBean definition

```
<mbean   code="org.jboss.ha.framework.server.ClusterPartition"
```

```
        name="jboss:service=DefaultPartition"/>
```

In the current version of the clustering, it is not possible to hot-deploy services based on HAPartition (but it is possible to hot-deploy new partitions and related services if performed at the same time).

For the HAPartition MBEAN, available attributes are:

Attribute	Mand-ated	Default value	Description
PartitionName	Opt.	DefaultPartition	the name of the partition (which define the cluster). Every node starting a partition with the same name form a cluster.
DeadlockDetection	Opt.	false	Boolean property that tells JGroups to run message deadlock detection algorithms with every request.
PartitionConfig	Opt.	See below	JGroups property string describing the stack of protocol to be used and their configuration.

The PartitionConfig attribute is an XML string that describes and configures the JGroups stack of protocols. For more information about writing your connection string, take a look at the JGroups documentation on http://www.jgroups.org/. The default connection string is:

Listing 3-2. JGroups protocol stack configuration

```
<Config>
  <!-- UDP: if you have a multihomed machine,
       set the bind_addr attribute to the appropriate NIC IP
       address -->
  <!-- UDP: On Windows machines, because of the media sense
       feature being broken with multicast (even after disabling
       media sense) set the loopback attribute to true -->
  <UDP mcast_addr="228.1.2.3" mcast_port="45566"
       ip_ttl="64" ip_mcast="true"
       mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
       ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
       loopback="false" />
  <PING timeout="2000" num_initial_members="3"
        up_thread="true" down_thread="true" />
  <MERGE2 min_interval="5000" max_interval="10000" />
  <FD shun="true" up_thread="true" down_thread="true" />
  <VERIFY_SUSPECT timeout="1500"
                  up_thread="true" down_thread="true" />
  <pbcast.NAKACK gc_lag="50"
       retransmit_timeout="300,600,1200,2400,4800"
                 up_thread="true" down_thread="true" />
  <UNICAST timeout="5000" window_size="100" min_threshold="10"
           down_thread="true" />
  <pbcast.STABLE desired_avg_gossip="20000"
                 up_thread="true" down_thread="true" />
  <FRAG frag_size="8192"
        down_thread="true" up_thread="true" />
  <pbcast.GMS join_timeout="5000" join_retry_timeout="2000"
              shun="true" print_local_addr="true" />
  <pbcast.STATE_TRANSFER up_thread="true" down_thread="true" />
</Config>
```

A Short description of the default protocols used by JBoss is given here:

- **UDP**: Transport, sends and receives unicast/multicast packets
    - ip_ttl (*4*): Time-to-live for IP Multicast packets.
    - ip_mcast (*true*): Use IP multicast. If set to false, JGroups will send *n* unicast packets rather then 1 multicast packet.

- mcast_send_buf_size (*150000*), mcast_recv_buf_size (*80000*), ucast_send_buf_size (*150000*), ucast_recv_buf_size (*80000*): Network buffer sizes for receiver DatagramSockets and sender MulticastSockets. The bigger the better, because JGroups can avoid packets dropped due to network buffer overflow.

- loopback (*true*): Place outgoing packets into the incoming queue (loopback). For unicasts, this means JGroups will not even put the packet on the network. For multicast packets, we send them, but discard our own incoming copy.

- PING: Initial (dirty) discovery of members. Used to detect the coordinator (oldest member). Each member 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 GMS). If nobody responds, we assume we are the first member of a group.

  - timeout (*2000*): Wait for 2 seconds…

  - num_initial_members="3": …or 3 responses, whichever comes first.

  - up_thread (*true*): Each protocol has an up-queue and a down-queue. When a message is received from above, it is placed into the down-queue. Each queue is handled by 1 thread, which continuously removes queue items (blocking until one is available), processes them and then passes them down. If up_thread="false", then we use the caller's thread, avoiding context switching.

  - down_thread (*true*): Same for down-queue thread.

- **MERGE2**: If a group gets split for some reasons (e.g. network partition), this protocol merges the subgroups back into one group. It is only run by the coordinator (=oldest member in a cluster), and periodically multicasts its presence. If another coordinator (for the same group) receives this message, it will initiate a merge process. Note that this merges subgroups {A,B} and {C,D,E} back into {A,B,C,D,E}, but it does not merge state. The application/service has to handle the viewChange(MergeView) callback to merge state.
    - min_interval (*5000*), max_interval (*10000*): Interval is a random number between 5 and 10 seconds.
- **FD**: Failure detection based on heartbeat messages. If reply is not received without timeout ms, max_tries times, a member is declared suspected, and will be excluded by GMS. If we use **FD_SOCK** instead, then we don't send heartbeats, but establish TCP sockets; and declare a member dead only when a socket is closed.
    - shun (*true*): Once a member is excluded from the group, and then rejoins (e.g. because it didn't crash, but was just slow, or a router that had crashed came back), it will be excluded (=shunned) and then has to rejoin. JGroups allows configuring itself such that shunning leads to automatic rejoins and state transfer (default in JBoss).
    - timeout (*2500*): Max number of ms to wait for 1 response.
    - max_tries (*5*): Max number of missed responses until a member is declared as suspected.
- **VERIFY_SUSPECT**: Verifies that a suspected member is really dead by pinging that member once again. Drops suspect message if member does respond. Tries to minimize false suspicions.

- **pbcast.NAKACK**: Lossless and FIFO delivery of multicast messages, using negative acks. E.g. when receiving P:1, P:3, P:4, a receiver asks P for retransmission of message 2.
    - gc_lag (*50*): always leaves 50 messages in the retransmit buffer.
    - retransmit_timeout (*300,600,1200,2400,4800*): Asks for retransmission of the **same** msg after 300ms, then 600ms, then 1200, etc.
    - max_xmit_size (*8192*): Retransmissions are bundled, but obviously cannot exceed the max fragmentation size (e.g. set in FRAG.frag_size). This value has to be <= to FRAG.frag_size.
    - use_mcast_xmit (*false*): Multicast retransmissions. If changes are that others are missing the same message, then this will reduce retransmissions.
- **pbcast.STABLE**: Garbage collects messages that have been seen by all members of a cluster. Each member has to store all messages because it may be asked to retransmit. Only when we are sure that all member have seen a message, can it be removed from the retransmission buffers. STABLE periodically gossips its highest and lowest messages seen. The lowest value is used to compute the min (all lowest seqnos for all members), and messages with a seqno below that min can safely be discarded.
    - desired_avg_gossip (*20000*): Gossip randomly every 20 secs.
- **UNICAST**: Lossless and FIFO delivery of unicast messages
    - Timeout (*600,1200,2400*): Asks for retransmission of the same msg after 600ms, then 1200ms, then 2400, etc.

- window_size (*100*): Max size of the sliding window. If this is exceeded, we block until we drop below min_threshold. Used for flow control.

- min_threshold (*10*): (see above)

- **FRAG**: Fragments messages larger than frag_size bytes. Unfragments at the receiver's side. Works for both unicast and multicast messages.

    - frag_size (*8192*): Max frag size in bytes. Messages larger than that are fragmented.

- **pbcast.GMS**: Group Membership Service. Responsible for joining/leaving members. Also handles suspected members, and excludes them from the membership. Sends Views (i.e. topology configuration) to all members when a membership change has occurred.

    - join_timeout (*5000*): Wait for 5 secs for a valid response until we retry the JOIN (sent to the coordinator)

    - join_retry_timeout (*2000*): If we have to retry the JOIN, wait 2 secs before retrying

    - shun (*true*): Shun a member that was declared dead, but came back nevertheless (see above). Member has to leave and rejoin

    - print_local_addr (*true*): Print the member's local address to stdout

- **pbcast.STATE_TRANSFER**: Allows a joining member to retrieve a shared group state from the oldest member (i.e. coordinator). Other members do not have to stop sending messages, while state transfer is in progress.

 If you are using a Windows server, you must modify the default connection string to workaround a Windows bug in the MediaSense feature and multicasting. In the above stack, modify the UDP protocol and set the **loopback** property to true.

 If you have a multi-homed machine (a machine with more than one network adapter), modify the UDP protocol of the above stack and set the **bind_addr** attribute to the IP address of the network adapter to be used.

 If you want to run several clusters on the same network, you can change the name of the partition so that different nodes use different partition names. However, this requires additional changes as many services will look for a partition named "DefaultPartition" and will fail to start if such a partition does not exist (or has been renamed). Consequently, it is much easier to leave the partition name as is and instead to change the multicast address being used ("mcast_addr" attribute of the UDP layer).

Chapter

4

4. HA-JNDI

Naming service and JBoss clustering

Having full-featured clustered EJBs is surely a good thing. Nevertheless, as almost any EJB access starts with a lookup of its home interface in a JNDI tree, the current clustering features would be almost useless without a clustered JNDI tree. Luckily, JBoss 3.0 provides such a feature. The current section will explains how it works and how lookups and bindings are resolved.

JNDI and HA-JNDI

There is a global, shared, cluster-wide JNDI Context that clients can use to lookup and bind objects. This is HA-JNDI. Remote Clients connecting to HA-JNDI will get fail-over and load-balancing. Things they bind to the HA-JNDI Context will also be replicated across the cluster so that if a node is down, the bound objects will still be available for lookup.

On the server side, new InitialContext(), will be bound to a 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.

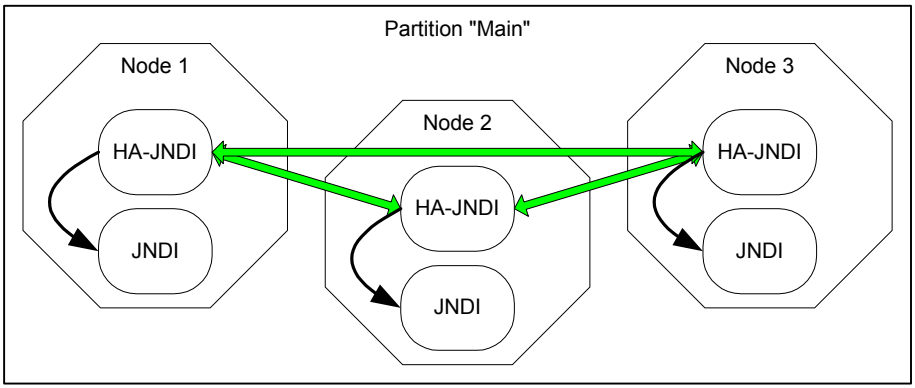


Figure 7. HA-JNDI name resolution

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. So, a EJB home lookup through HA-JNDI, will always be delegated to the local JNDI instance.

In the current implementation, HA-JNDI works with its own cluster-wide JNDI tree as well as the local JNDI tree available in JBoss. Please 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. Nothing prevents you though of using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

HA-JNDI set up

To set up a HA-JNDI service, you need to insert a XML entry in your JBoss configuration file (either in jboss-services.xml or any configuration file in the /deploy folder. The cluster-service.xml file already includes such a definition. You can see that this MBean depends on the DefaultPartition MBean defined above it.

Listing 4-1. HA-JNDI MBean definition

```
<mbean code="org.jboss.ha.jndi.HANamingService"
```

```
         name="jboss:service=HAJNDI">
      <depends>jboss:service=DefaultPartition</depends>
   </mbean>
```

Available attributes are:

Attribute	Mandated	Default value	Description
PartitionName	Optional	DefaultPartition	the name of the partition (which define the cluster) that must be used to make communicate the different replicated instances of the HA-JNDI service
BindAddress	Optional		Address to which the HA-JNDI server will bind waiting for JNP clients. Only useful for multi-homed computers.
Port	Optional	1100	Port to which the HA-JNDI server will bind waiting for JNP clients.
Backlog	Optional	50	Backlog value used for the TCP server socket waiting for JNP clients.

RmiPort	Optional	0	Once the JNP client has downloaded the stub for the HA-JNDI server, they use a standard RMI connection to communicate together. This attribute can be used to determine which port the server should use.
AutoDiscoveryAddress	Optional	230.0.0.4	Multicast address to listen to for JNDI automatic discovery.
AutoDiscoveryGroup	Optional	1102	Multicast group to listen to for JNDI automatic discovery.

Consequently, only the PartitionName attribute differentiates the JNDI service from the HA-JNDI service. In most cases, no attribute needs to be set.

So, if you wanted to hook up HA-JNDI to the example partition you set up in 1.D and change the binding port, the Mbean descriptor would look as follows.

Listing 4-2. Overriding HA-JNDI default values

```
<mbean code="org.jboss.ha.jndi.HANamingService"
  name="jboss:service=HAJNDI">
  <depends>jboss:service=MySpecialPartition</depends>
  <attribute name=”PartitionName”>MySpecialPartition</attribute>
```

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

HA-JNDI binding and lookup rules

As introduced before, there is a tight coupling between the JNDI and HA-JNDI services on each node of the cluster.

There are no special interfaces when working with HA-JNDI. When a client does a lookup, it is transparent to the client whether the bounded object lives in the global JNDI tree or the local JNDI tree.

Consequently, when a client lookups a name through a HA-JNDI service hosted on node N, 3 cases can arise:

1. the binding has been made through HA-JNDI and is available cluster-wide
2. the binding has been made through local JNDI on node N
3. the binding has been made through a local JNDI residing on a different node.

Here are the steps performed by the HA-JNDI service when it receives a lookup query (1):

1. If the binding is available in the cluster-wide JNDI tree and it returns it. (1.1)
2. If the binding is not in the cluster-wide tree, it delegates the lookup query to the local JNDI service and returns the received answer if available. (1.2)

3. If not available, the HA-JNDI services asks all other nodes in the cluster (1.3 and 1.3') if their local JNDI service (1.4 and 1.4') owns such a binding and returns the an answer from the set it receives.

4. If no local JNDI service owns such a binding, a NameNotFoundException is finally raised.

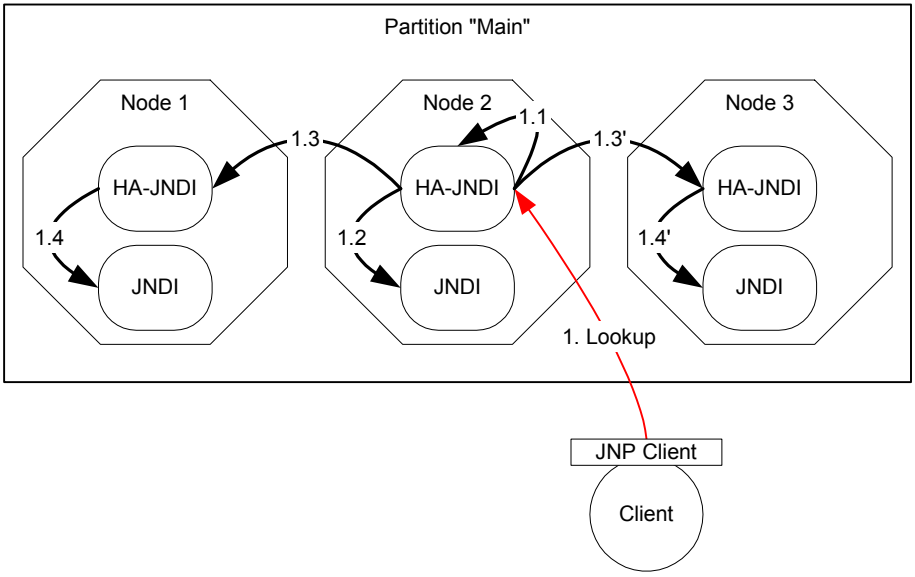


Figure 8. HA-JNDI detailed lookup resolution process

What needs to be remembered from this scheme is that:

- 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 higher 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.

- Using a node's local JNDI to lookup a name that has been bound through HA-JNDI will raise a NameNotFoundException exception: HA-JNDI has its own cluster-wide JNDI tree. If you want to access HA-JNDI from the server side within code, you must explicitly get an InitialContext by passing in JNDI properties.

Listing 4-3. Setting JNDI properties in code to access 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);
```

- As EJB containers bind their home stubs using the local JNDI service:

    1. Home stubs are available cluster-wide

    2. If different beans (even of the same type, but participating in different partitions) 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!

- It is possible to start several HA-JNDI services that use different partitions. This can be used, for example, if a node is part of

many clusters/partitions. In this case, take care of setting a different port or IP address for both services.

HA-JNDI Design Note

This is a short section, but the reasons why there is one global HA-JNDI cluster-wide context and a local JNDI context on each cluster node is as follows:

- We didn't want any migration issues with applications already assuming that their JNDI implementation was local. We wanted clustering to work out-of-the-box with just a few tweaks of configuration files.
- We needed a clean distinction between locally bound objects and cluster-wide objects.
- In a homogeneous cluster, this configuration actually cuts down on the amount of network traffic.
- 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/create bindings.

HA-JNDI client and auto-discovery

When working with a single JBoss server, the JNDI configuration of the clients is simplified: they only need to know the hostname/IP address and port number of the remote JNDI server.

When working with a cluster of JBoss servers, the configuration is not that easy. Which server will be running when our clients will start? If we had this kind of certitude, clustering would not be that much necessary. Consequently, the java.naming.provider.url JNDI setting can now accept a list of urls separated by a comma. Example:

Listing 4-4. Sample HA-JNDI property string for multiple known servers

```
java.naming.provier.url=server1:1100,server2:1100,server3:1100,server
4:1100
```

When initialising, the JNP client code will try to get in touch with each server from the list, one after the other, stopping as soon as one server has been reached. It will then download the HA-JNDI server stub from this host. The downloaded smart stub contains the logic to fail-over to another server if necessary and the updated list of currently running servers. Furthermore, each time a JNDI invocation is made to the server, the list of target is updated (only if the list has changed since the last call).

Note that this feature is also available for the standard JNDI server. But in this case, the downloaded smart stub does not know how to fail-over and does not update its view of the cluster (it simply ignores the notion of cluster).

In highly dynamic environments, where many servers start, stop and are moved, this solution can still be frustrating to configure. Consequently, a new feature, called “auto-discovery”, has been set up. If the property string java.naming.provier.url is empty or if all servers it mentions are not reachable, the JNP client will try to discover a bootstrap HA-JNDI server through a multicast call on the network. Thus, if you have a LAN or if your WAN is configured to propagate such multicast datagrams, the client will be able to get a valid HA-JNDI server without any configuration.

The auto-discovery feature uses multicast group address 230.0.0.4:1102.

This feature is not available with the standard JNDI server (only the HA-JNDI server).

HA-JNDI JNP specific properties

When you create a new InitialContext, you can initialize it with a set of properties. By default, these properties will identify which JNDI Service Provider to use, where is located the target server (see java.naming.provier.url above), etc.

When using HA-JNDI in a clustered environment you can specify a set of other properties as well:

Property	Description
`jnp.disableDiscovery`	When set to "true", this property disables the automatic discovery feature presented in the previous section. Default is "false".
`jnp.partitionName`	In an environment where multiple HA-JNDI services, bound to distinct HAPartitions are started, this property allows you to configure which partition you are looking for when the automatic discovery feature is used. If you do not use the automatic discovery feature (because you explicitly provide a list of valid provider in java.naming.provier.url for example), this property will not be used. When not set (default), the automatic discovery will select the first HA-JNDI server that responds, independently of its partition name.
`jnp.discoveryTimeout`	Determines how much time the context will wait for a response to its automatic discovery packet. Defaults to 5000ms.
`jnp.discoveryGroup`	Determine which multicast group address is used for the automatic discovery feature. Defaults to

	230.0.0.4
`jnp.discoveryPort`	Determine which multicast group port is used for the automatic discovery feature. Defaults to 1102

Chapter 5

5. Clustering EJB

Clustering entity and session beans

This section presents how to cluster the different kind of entity beans and the configuration settings that apply.

Bean clustering requires at least JDK 1.3. JDK 1.2.2 or below is not supported.

While clustering your application, you should keep in mind the following guidelines:

- If your application is composed of many different beans, deploy them all on all nodes. While it is possible to split the beans on different nodes (B1 and B2 on Node 1 and 2 and B3 on node 2 and 3 for example), this may have a serious, negative performance impact. Furthermore, if your JBoss instances are not configured to use the distributed transaction manager, your calls are very likely to fail.

- Simplest is better: while complicated cluster topology can be built (with nodes participating in many clusters for example), keeping it simple avoids to fall in an administrative nightmare.

Stateless Session Beans

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

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

Listing 5-1. Setting a stateless session bean as clustered

```
<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>
```

In the bean configuration, only the <*clustered*> tag is mandatory to indicate that the bean will work clustered. All other elements (sub-elements of <*cluster-config*>) are optional and their default values are indicated in the sample configuration above.

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

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

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

Stateful Session Beans

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

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

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

Listing 5-2. Session State MBean definition

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

Available attributes are:

Attribute	Mand-ated	Default value	Description
JndiName	Opt.	/HAPartition/Default	The JNDI name under which this HASessionState will be bound
PartitionName	Opt.	DefaultPartition	Name of the partition in which the current HASessionState protocol will work.
BeanCleaningDelay	Opt.	30*60*1000 (30 minutes)	Number of miliseconds after which the HASessionState 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.

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

Listing 5-3. Setting a stateful session bean as clustered

```
<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 will work clustered. All other elements (sub-elements of *<cluster-config>*) are optional and their default 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. By default it uses the default HASessionState.

The description of the remaining tags is identical to the one for stateless session bean.

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

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

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

Before replicating your bean, the container will detect if your bean implements this method and possibly call it. If the bean has not bean modified (or not enough to require replication, depending on your own preferences), the replication will not occur.[>= 3.0.1 only]

Entity Beans

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

Listing 5-4. Setting an entity bean as clustered

```
<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>
```

In the entity bean configuration, only the <*clustered*> tag is mandatory to indicate that the bean will work clustered. All other elements (sub-elements of <*cluster-config*>) are optional and their default values are indicated in the sample configuration above.

The description of the remaining tags is identical to the one for stateless session bean.

Entity Synchronization

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

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. Also, the JBoss CMP 1.0 (JAWS) and CMP 2.0 implementations have row-locking capabilities under the <*row-locking*> feature. See CMP documentation for more details.

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.)

Message Driven Beans

No features currently available.

Load-balance Policies

For each bean, you can decide, for your home and remote proxies, which load-balancing policy you want to use. This section present the policies.

JBoss 3.0.x

Two policies are available by default:

- Round-Robin (org.jboss.ha.framework.interfaces.RoundRobin): each call is dispatched to a new node.
- First Available (org.jboss.ha.framework.interfaces.FirstAvailable): one of the available target nodes is elected as the main target and is used for every call: this elected member is **randomly** chosen from the list of targets. When the list of target nodes changes (because a node starts or dies), the policy will re-elect a target node **unless** the currently elected node is still available. Each proxy elects its own target node independently of the other proxies.

In JBoss 3.0.x, each proxy (home or remote) has its own list of available target nodes. Consequently, some side-effects can occur.

For example, if you cache your home proxy and re-create a remote proxy for a stateless session bean (with the Round-Robin policy) each time you need to make an invocation, a new proxy, containing the list of available targets), will be downloaded for each new remote proxy. Consequently, as the first target node is always the first in the list, calls will not seemed to be load-balanced because there is no usage-history between different proxies.

Listing 5-5. Recreating a new remote proxy for each call

```
…
Home myHome = …;
while (jobToDo)
{
  Remote myRemote = myHome.create (); // get a brand new proxy and
                                      // its own list of
                                      // target nodes
  myRemote.doTheJob (…); // load-balance calls starting at position 1
                         // => calls are not load-balanced because
                         //    only one call
                         //    is made on each remote proxy
}
…
```

Listing 5-6 Reusing a remote proxy for each call

```
…
Home myHome = …;
Remote myRemote = myHome.create (); // get a brand new proxy and
                                    // its own list of target nodes
while (jobToDo)
{
  myRemote.doTheJob (…); // load-balance calls starting at position 1
                         // => calls are load-balanced because each
                         //    call will use the next available
                         //    target
}
…
```

JBoss ≥ 3.2

Three policies are available by default:

- Round-Robin (org.jboss.ha.framework.interfaces.RoundRobin): each call is dispatched to a new node. The first 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 used for every call. The elected member is **randomly** chosen from the list of targets. When the list of target nodes changes (because a node starts or dies), the policy will re-elect a target node **only** if the currently elected node is no longer available. Each proxy elects its own target node.

- First AvailableIdenticalAllProxies (org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies): Same behaviour as the First-Available policy but the elected target node is shared by all proxies of the same "family" (see below for more information).

In JBoss 3.2 (and upper), the notion of "Proxy Family" is defined. A **Proxy Family** is a set of proxies that all make invocations against the same replicated target. For EJBs for example, all proxies targeting the same EJB in a given cluster are said to be of the *same proxy family*. Note that home and remote proxies of a given EJB are in two different families.

All proxies of a given family share the same list of target nodes (plus some other information such as the view id, etc.) in a structure called *FamilyClusterInfo*. This structure can also contain some arbitrary information stored by the proxies themselves or their associated load-balancing policy. Thus each proxy (home or remote) has a means to

share information with other proxies of the same family. Furthermore, if the cluster topology for a given EJB changes, the new list of target nodes will only be refreshed once independently of the number of instantiated proxies (Collection of entity beans returned from a finder for example).

These changes will remove the side-effects that are possible with the solution implemented in JBoss 3.0. Thus, the code in Listing 5-5 that continuously re-creates remote proxies for each invocation will correctly load-balance calls with JBoss 3.2 (and upper).

Chapter 6

6. HTTP Session clustering

Clustering HTTP Sessions with Tomcat

Introduction

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:

1. Session state replication
2. Load-balance of incoming invocations

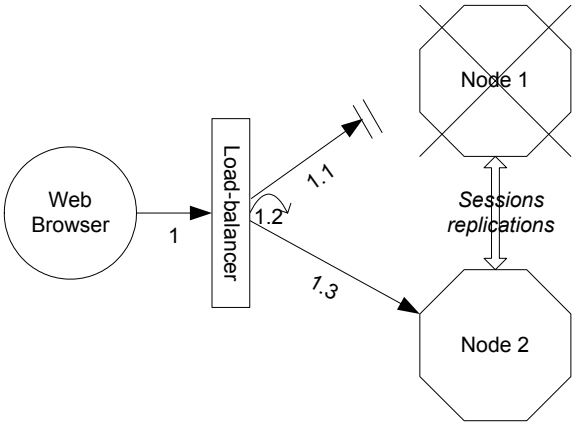


Figure 9HTTP Session failover

State replication is directly handled by JBoss whereas load balancing of invocations requires additional software or hardware.

As load-balancing is not handled by JBoss itself, this chapter will not specifically focus on calls load-balancing. Nevertheless, as this is a very common scenario, we will demonstrate how to setup Apache and mod_jk. This activity could be either performed by specialized hardware switches or routers (Cisco LoadDirector for example) or any other dedicated software[4] though.

In the figure above, a load-balancer tracks the HTTP requests and, depending on the session to which is linked the request, it dispatches the request to the appropriate node. This is called a load-balancer with **sticky-sessions:** once a session is created on a node, every future request will also be processed by the same node. This is an important feature as web applications may make concurrent calls to the web server (HTML frames for example). Thus, if all concurrent requests do not go to the same node, each node may concurrently modify the same

[4] A JBoss load-balancer has also been implemented by Thomas Peuss, it is not described here though..

data thus breaking the coherency of the HTTP session (the same HTTP session may be different on each node).

Do you really need HTTP Sessions replication?

As you saw in the previous section, requests load-balancing and state replication are two different issues. Consequently, you can use a load-balancer without replicating your sessions.

Using a load-balancer that supports sticky-sessions without replicating the sessions allows you to scale very well without the cost of session state replication: each query will always be handled by the same node. But in the case a node dies, the state of all client sessions hosted by this node are lost (the shopping carts for example) 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 the database. In other situations, loosing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

HTTP Session Replication Setup

Introduction

In the following sections, we will follow the complete trip of an HTTP invocation and sequentially show how to setup:

- Apache and mod_jk to act as a front-end load-balancer,
    - HTTP Session Replication in Tomcat 4.x and 5.x
- Activate and tune session replication in your Web Application

Apache and mod_jk

Apache[5] is a well-known web server which can be extended by plugging modules. One of these modules, mod_jk has been specifically designed to forward 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 an association list to remember which specific node of the cluster has served which specific client (i.e. "sticky sessions"), and this is what is actually interesting for us[6].

The following sections will guide you through all steps required to install and configure mod_jk:

- Download and install mod_jk binaries
- Configure Apache to load the mod_jk module
- Configure the mod_jk module
- Configure the set of Servlet containers to which mod_jk will forward HTTP requests
- Configure each Tomcat instance to have a distinct name

In the following section, we show the configuration steps for **mod_jk**.While it is also possible to use the newest mod_jk2 in conjunction with JBoss clustering, configuration files are not the same. Furthermore, specific care must be taken when selecting an Apache 2.x Threading Library under a non-threaded OS (Linux 2.4.x for example): if you are using a "process forking" model, as mod_jk2 sticky-session information is not stored in shared memory, it is lost

[5] Apache runs on lots of different OS/platforms. It can be download here: http://httpd.apache.org/

[6] mod_jk also exists for IIS, Domino and Netscape web servers

each time a process is spawn/dead and this leads to a non-consistent load-balancing/sticky-sessions behaviour.

Download and install mod_jk binaries

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. In this chapter, we have used Apache **version 1.3.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 will **use mod_jk 1.2**. It can be downloaded from

http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/

Under Windows, it will be a .DLL file, whereas under Linux and Solaris it will be a .SO fill. Most of the time, the filename will include its version number ("mod_jk_1.2.5_1.3.27.dll"). To finish this step, **rename it** "mod_jk.dll" or "mod_jk.so" (to remove the version number) and **copy this file under APACHE_HOME/modules/**.

Configure Apache to load the mod_jk module

In this section, we will tell Apache to load the mod_jk module and set mod_jk's basic settings.

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

Listing 6-1. Including mod_jk's configuration file in Apache's main configuration file (conf/httpd.conf)

```
...
```

```
#<VirtualHost *>
#    ServerAdmin webmaster@dummy-host.example.com
#    DocumentRoot /www/docs/dummy-host.example.com
#    ServerName dummy-host.example.com
#    ErrorLog logs/dummy-host.example.com-error_log
#    CustomLog logs/dummy-host.example.com-access_log common
#</VirtualHost>

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

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

Listing 6-2. mod_jk's configuration file (conf/mod-jk.conf)

```
# Load mod_jk module.
# Specify the filename of the mod_jk
# lib you've downloaded and  installed in the previous section

LoadModule jk_module modules/mod_jk.dll

# 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 indicate to send SSL KEY SIZE,
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories

# JkRequestLogFormat set the request format
JkRequestLogFormat "%w %V %T"

JkMount /* loadbalancer
```

```
# if you wanted to only load-balance a sub-context, you could
# map the module differently, such as:
# JkMount /myContext/* loadbalancer
```

Two settings are very important:

1. The first one, “LoadModule”, 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. That is the reason why we have renamed that file: to make it simply to reference it from the configuration file.

2. The last one, “JkMount”, 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 will be sent to the mod_jk load-balancer. That is fine if Apache is only use as a load-balancer. However, if you plan to use Apache to serve static content as well, you would only send some specific URLs to mod_jk.

You will most probably not change the other settings: they are used to tell mod_jk where to put its logging file, which logging level to use, etc.

Configure the worker nodes

At this point, Apache is configured to load mod_jk at startup and to let it handle HTTP requests that match a given URL criteria. We still need to tell mod_jk to which JBoss node these requests must be sent to.

In this section we will configure mod_jk workers file: **conf/workers.properties**. This file specify where are located the different Servlet containers 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 would look like this:

Listing 6-3. conf/workers.properties sample file for 2 workers Servlet container

```
# Define the first node...
worker.node1.port=8009
worker.node1.host=node1.mycompany.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.local_worker=1
worker.node1.cachesize=10

# ...and the second node.
worker.node2.port=8009
worker.node2.host=node2.mycompany.com
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.local_worker=1
worker.node2.cachesize=10

# Now we define the load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=node1, node2
worker.loadbalancer.sticky_session=1
worker.loadbalancer.local_worker_only=1
worker.list=loadbalancer
```

First, each node is defined using the worker.XXX naming convention where XXX represents an arbitrary name you choose for one of the target Servlet container. For each worker, you must give the host name (or IP address) and port number of the AJP13 connector running in the Servlet container.

> AJP13 is a protocol used between mod_jk and the servlet container to replace straight HTTP and is more resource-efficient than straight HTTP. Other

protocols are also available for mod_jk to Servlet container communication. See the mod_jk configuration for more information[7].

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, the more HTTP requests it 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**.

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.

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).

Activating and configuring Tomcat Session Replication Service

Three steps are required to configure Tomcat:

1. Configure Tomcat snapshotting mode

   This step is **optional** as default settings already exist.

[7] http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/

2. Configure Tomcat mod_jk connector

   This step is **optional** as default settings already exist.

3. Name each Tomcat engine of the cluster

   This last step is mandatory.

To activate the HTTP session replication service, you must first have a correctly running cluster. The simplest way is to start JBoss using the "all" configuration:

```
     run.bat -c all
or
     ./run.sh -c all
```

Specifically, Tomcat HTTP session replication requires the clustering files as well as the jbossha-httpsession.sar file in /deploy. As this file is already part of the "all" configuration, running this configuration is enough to enable Tomcat HTTP session replication: no additional step is required.

> HTTP Sessions replication for Tomcat is handled by a JBoss service that is independent of the Servlet container. The Servlet container in itself doesn't contain any replication code but, instead, relies on this service.

Configuring Tomcat Snapshotting Mode

Once a session has been modified, it must be replicated to other nodes. However, it is possible to determine what triggers the replication of the sessions. This step is optional as default configuration settings already exist.

Take a look at the Tomcat configuration in deploy/jbossweb-tomcat41.sar/META-INF/jboss-service.xml or, if you are using Tomcat 5.0, deploy/jbossweb-tomcat50.sar/META-INF/jboss-service.xml:

```
…
<server>
…
  <mbean code="org.jboss.web.tomcat.tc5.Tomcat5"
    name="jboss.web:service=WebServer">
…
    <attribute name="SnapshotMode">instant</attribute>
    <!-- you may switch to "interval" -->
    <attribute name="SnapshotInterval">2000</attribute>
    …
```

The first element is the **SnapshotMode** attribute. It defines when modified sessions should be replicated to the other nodes of the cluster. By default, Tomcat HTTP session replication uses a method called "instant snapshotting": session replication is initiated directly after each HTTP request. To use instant-snapshotting, set the **SnapshotMode** attribute to "**instant**" (in this case, the "SnapshotInterval" attribute is ignored).

Another snapshotting scheme is available; it is called "interval snapshotting". In this case, sessions are collected for a given period of time and then replicated en-bloc. This may give you some performance when you work with HTML-framesets where concurrent requests for the same ID can occur. Without interval snapshotting every request triggers replication. To use interval-snapshotting, set the **SnapshotMode** attribute to "**interval**" and "**SnapshotInterval**" to the interval duration in milliseconds.

Configuring Tomcat mod_jk Connector

Requests between Apache and Tomcat are transmitted using a specific protocol: mod_jk. Consequently, Tomcat needs to start and additional connector to accept mod_jk requests. As the Tomcat mod_jk connector is already started by default, this configuration step is optional.

If you are using Tomcat 4.1.x, the configuration can be found in JBOSS_HOME/server/*all*/deploy/jbossweb-tomcat41.sar/META-INF/jboss-service.xml:

```
...
    <!-- A AJP 1.3 Connector on port 8009 -->
    <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
       address="${jboss.bind.address}" port="8009"
       minProcessors="5" maxProcessors="75"
       enableLookups="true" redirectPort="8443"
       acceptCount="10" debug="0" connectionTimeout="20000"
       useURIValidationHack="false"
       protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"/>
...
```

If you are using Tomca 5.0.x, the configuration can be found in file JBOSS_HOME/server/*all*/deploy/jbossweb-tomcat50.sar/server.xml:

```
...
   <!-- A AJP 1.3 Connector on port 8009 -->
   <Connector port="8009" address="${jboss.bind.address}"
        enableLookups="false" redirectPort="8443" debug="0"
        protocol="AJP/1.3" />
...
```

The minProcessors and maxProcessors attributes of Tomcat 4.1.x define the minimum and maximum size of the thread pools that will accepts incoming query.

This value of the **maxProcessors** attribute must at least match the value of the mod_jk thread pool defined for this worker (worker.nodeXXX.cachesize key in the APACHE_HOME/conf/workers.properties file)

Name each Tomcat Instance of the Cluster

Each Tomcat instance that will be participate in the cluster now needs to be named accordingly to the names we have used in the workers.properties file ("node1" and "node2" in our example).

If you are using **Tomcat 4.1.x**, edit the file JBOSS_HOME/server/*all*/deploy/ jbossweb-tomcat41.sar/META-

INF/jboss-service.xml and add a "**jvmRoute**" attribute to the Tomcat engine configuration element:

```
...
<attribute name="Config">
   <Server>
      <Service name="JBoss-Tomcat">
         <Engine name="MainEngine" defaultHost="localhost"
                 jvmRoute="node1">
            <Logger className="org.jboss.web.tomcat.Log4jLogger"
...
```

If you are using **Tomcat 5.0.x**, edit the file JBOSS_HOME/server/*all*/deploy/ jbossweb-tomcat50.sar/server.xml and add a "**jvmRoute**" attribute to the Tomcat engine configuration element:

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

   <Logger className="org.jboss.web.tomcat.Log4jLogger"
...
```

It is very important that the names of the nodes known by Apache mod_jk in worker.properties **exactly match** the names of the "jvmRoute" parameter in the Tomcat configuration!

In this example we have modified the Tomcat configuration files in the "all" sub-directory, the only default configuration which has JBoss clustering enabled by default.

Activate session replication in your Web Application

Once your load-balancer and your Servlet container are configured to replicate HTTP sessions, the last element that must be configured is … your web application itself!

This can be done by declaring a `distributable` tag at the beginning of the `WEB-INF/web.xml` file of your WAR. This tag has no parameter, it is just some kind of placeholder that tells the Servlet container that session should be replicated:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <distributable/>
   …
</web-app>
```

That's all! You can now test HTTP session clustering.

Session Replication Tuning

(This is only available since JBoss 3.2.3)

Depending on what your requirements are, it is possible to tune the replication settings of each clustered Web application. This is achieved through the META-INF/jboss-web.xml configuration of your WAR archive.

```
<?xml version="1.0"?>
<!DOCTYPE jboss-web
    PUBLIC "-//JBoss//DTD Web Application 2.3V2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_3_2.dtd"

<jboss-web>
   ...
```

```
   <replication-config>

      <replication-trigger>
         SET_AND_NON_PRIMITIVE_GET
      </replication-trigger>

      <replication-type>SYNC</replication-type>

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

Replication Trigger

The first tag, “replication-trigger”, determines when should Tomcat consider that an attribute has been modified. One of the difficulties with HTTP sessions is that you can modify their content without actually calling setAttribute, for example:

```
   ...
   Object o = httpSession.getAttribute (“myHashtable”);
   java.util.Hashtable table = (java.util.Hashtable)o;

   table.put ("some key", "Updated Value");
   ...
```

The Servlet specification says nothing about that situation. Consequently, we would potentially have to replicate an HTTP Session each time a single “getAttribute” is performed, even if no state has been modified. However, if you know specifically how your application works, you can optimize the replication job by telling JBoss what should be considered as a modification of the content of the session. The possible values are:

1. "SET_AND_GET"
2. "SET_AND_NON_PRIMITIVE_GET" (default value)
3. "SET"

The first option is conservative but not optimal: it will replicate the session even if its content has not been modified but simply accessed by a call to getAttribute.

The second option is conservative but will only replicate if a non-primitive Object has been accessed (Integer, Long, String, etc.) because they are immutable. It is the default value.

The third option considers that the developer will explicitly call setAttribute on the session if it has to be replicated. The option can be enabled if the developer can insure JBoss that no indirect modification of the state will take place, this is the most efficient setting.

Replication Type

By default, JBoss will generate a synchronous call to replicate sessions. This means that each time an HTTP response is received by a client, its session has already been replicated.

If such a synchronous behaviour is not necessary, it is possible to reduce the latency perceived by the client by initiating the session replication but not waiting for the replication to terminate. Thus, the time is takes to replicate the session is not part of the request but is instead spent in parallel.

Chapter 7

7. Farming

Distributed Hot-Deployment

With JBoss clustering you can hot-deploy across the whole cluster just by plopping your EAR, WAR, or JAR into the deploy directory of one clustered JBoss instance. Hot-deploying on one machine will cause that component to be hot-deployed on all instances within the cluster.

Farming is enabled by default in the "all" configuration, so you will not have to set it up yourself. If you want to do it yourself though, simply create the XML file shown below and copy it to the JBoss deploy directory *$JBOSS_HOME/server/all/deploy/deploy.last/*.

farm-service.xml

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

<server>

 <classpath codebase="lib" archives="jbossha.jar"/>

 <mbean code="org.jboss.ha.framework.server.FarmMemberService"
   name="jboss:service=FarmMember,partition=DefaultPartition">
   ...
     <attribute name="PartitionName">DefaultPartition</attribute>
     <attribute name="ScanPeriod">5000</attribute>
     <attribute name="URLs">farm/</attribute>
   </mbean>
```

```
</server>
```

After deploying *farm-service.xml* you are ready to rumble. Here are the available attributes for configuring a farm:

Attribute	Mandated	Default value	Description
PartitionName	Required	DefaultPartition	Name of the cluster partition that this deployed farm belongs to.
URLs	Required	farm/	Directory where deployer watches for files to be deployed. This MBean will create the directory if it doesn't already exist. Also "." pertains to the configuration directory, i.e. *$JBOSS_HOME/server/all*
ScanPeriod	Required	5000	Interval at which the folder must be scanned for changes.

As the Farming service is an extension of the URLDeploymentScanner that scans deployments in /deploy, all other attributes have the same meaning. Check in the JBoss administration guide for more information.

Chapter 8

8. Cache Invalidation

Automatically Invalidating Specific Cache Entries

This chapter describes the JBoss Cache Invalidation Framework (CIF) that is available since JBoss 3.2. The framework allows you to link several caches that represent the same underlying data so that when one of the entities is modified, all concerned caches flush this particular identity. This mechanism allows them to cache only up-to-date information and very quickly invalidate any stale data.

Overview

The cache is a well-known and important player of the EJB specification for entity beans. It allows the container to keep a set of entity beans associated with an identity so that future access to the same bean will avoid a costly access to the database and the creation and initialisation of a new entity bean.

To determine the behaviour of caches, the EJB specification specifies 3 "Commit Options":

- Commit Option A: "I own the DB". With this setting, you tell the container that it is the only actor that will modify data in the

persistent store. Consequently, it can cache as many data as it wants because it will always be up-to-date (no other actor is allowed to make modifications without going through this particular container)

- Commit Option B or C[8]: "I don't own the DB". With this setting, you tell the container that other actors could potentially modify data without going through this container. They will not necessarily do this, but they may. Consequently, the container is not allowed to cache any data across transactions.

JBoss provides a fourth (proprietary) Commit Option: D. This level works like commit option A but with a timeout value: once the timeout has elapsed (for example 30 seconds), the instance is considered as invalid and removed from the cache. This scheme can be very useful when data does not change frequently or when it is not critical to work with really up-to-date data.

Consequently, to benefit from a cache, Commit Options A or D must be used. The problem is that Option A can only be very rarely used in production environment and Option D does not fit all needs: stale data can be used until the timeout has elapsed and non-staled data may be removed from cache (which is not efficient).

To overcome these restrictions, the Cache Invalidation Framework can be used in conjunction with Commit Option A: data are kept in cache and only removed when another actor (which may be another container, another JBoss instance or any other software) invalidates a given identity.

---

[8] The slight difference that exists between Commit Options B and C is not really interesting in this case. Commit Option C forces the container to definitively remove instances from the cache when the transactions commits/rollbacks while in Commit Option B, the container may simply flag them as invalid. This difference can for example be used to implement Optimistic Locking schemes.

Framework Architecture

The Cache Invalidation Framework is not tight to a specific cache implementation or invalidator. Instead, it is much more an "invalidation-router" that can route invalidation messages from specific invalidators to the appropriate(s) invalidation listeners (such as caches).

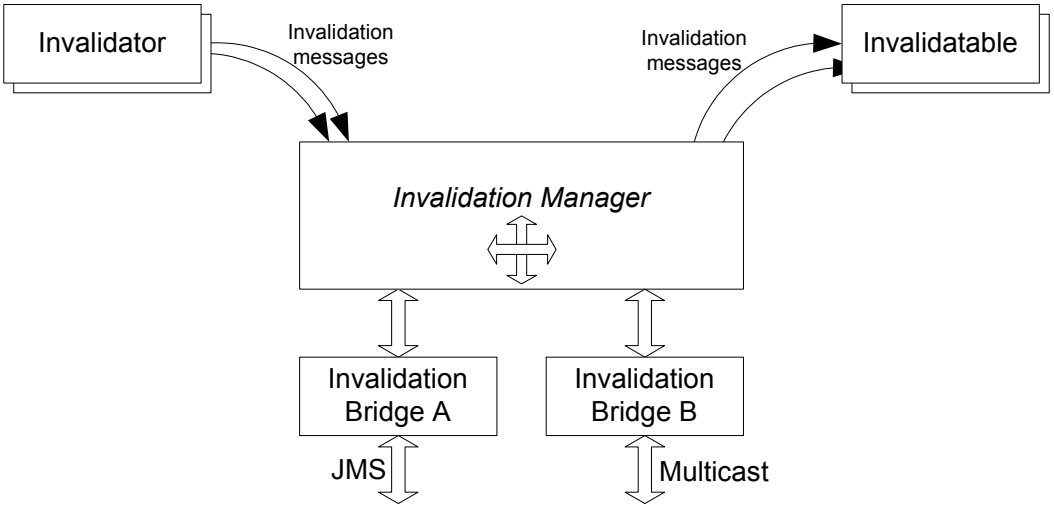


Figure 10 Cache Invalidation Framework Architecture

Invalidator and Invalidation listeners all work in the focus of a given **Invalidation Group**. An Invalidation Group gather all invalidations for a given domain (a specific entity bean for example). Thus, if we have two distinct entity beans (Customers and Orders) whose caches plug in the Invalidation Manager, two distinct Invalidation Groups are formed.

While more than one Invalidation Manager could run inside the same JVM, having a single manager per JVM will fit most if not all needs. Until now, everything that has been described happens inside the same virtual machine. To allow more than one JBoss instance to exchange invalidation messages, Invalidation Bridges must be used.

An Invalidation Bridge is a protocol specific component that plugs in an invalidation manager and can:

- Mass-receive invalidation messages for one or more Invalidation Groups
- Mass-send invalidation message for one or more Invalidation Groups

Consequently, the Cache Invalidation Framework can either work in pure single-JVM environments or in distributed/clustered environments: the Invalidation Manager doesn't care. It is up to the JBoss administrator to (hot-)deploy the appropriate Invalidation Bridges.

Thus, the above architecture is fully generic and not tied to a specific usage. The next section will describe the integration with EJB containers.

A default Invalidation Manager is started as part of the "default" and "all" configurations:

deploy/cache-invalidation-service.xml

```
<server>
...
  <mbean code="org.jboss.cache.invalidation.InvalidationManager"
         name="jboss.cache:service=InvalidationManager">
  </mbean>
...
<server>
```

The Invalidation Manager can perform both asynchronous and synchronous invalidations. By default, it will perform synchronous

invalidations. This behaviour can be change on an invalidation-messages basis when sending invalidation messages.

What the framework is not

People frequently confuse the following:

- Distributed cache invalidation
- Distributed cache
- Distributed locking
- Distributed transactions

While all these can be related, they are not equal.

A distributed cache keeps a consistent cache across a set of nodes and modifications done on a given node are generally replicated to other nodes. Furthermore, some implementations can be temporarily disconnected from the network and work from cache directly. Distributed caches are frequently provided with a distributed locking mechanism.

Locking is used with entity beans to solve one or both of the following requirements: instance (non-) concurrency and transaction isolation. When more than one client access a given entity bean, they same bean instance can only be accessed by a single “transaction” and each client must have its own view of the instance (isolation). This can be achieved through Pessimistic Locking for example: as long as one transaction uses a given instance, no other transaction can use it. To insure that only a single transaction has exclusive access to a given instance, a synchronization point must be used. Most of the time, this point is the database itself. Nevertheless, sometimes a distributed locking

mechanism is used at the application server directly: containers never directly lock the data in the database itself but instead lock beans at the application server level through a Distributed Lock Manager.

Distributed Transactions are used when a given transaction execution spans more than one node[9]. In this case, the transaction context must be transported to other nodes and all nodes participate in the same transaction. The keywords in this context are frequently: XA transactions (and XA drivers) and 2-Phase-Commits (2PC).

The current invalidation framework does not provide distributed caching, or distributed locking, or distributed transactions. Consequently, make sure your application does not required distributed locking.

EJB Integration

To integrate Entity Beans in the Cache Invalidation Framework, to main components have been added.

First, a extended version of JBoss Entity Bean cache must be used. This implementation knows how to register in the Invalidation Framework, listen to invalidation messages for a given Invalidation Group and invalidate these entries from the cache.

[9] This definition is not accurate but fits in our case

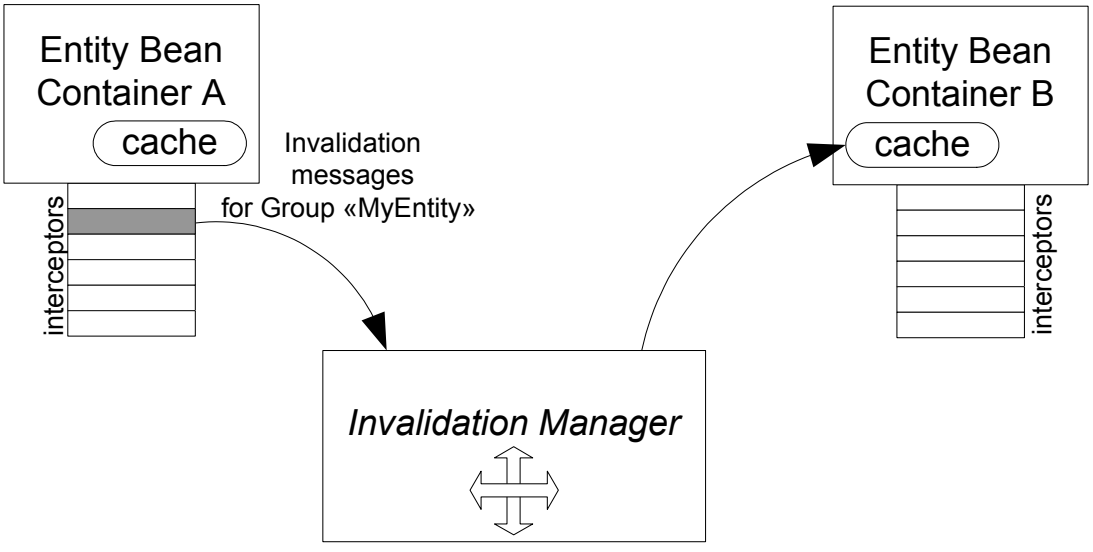


Figure 11Cache Invalidation Framework Integration in EJB

Second, a new Interceptor must be added to the container. This interceptor will detect when a specific bean has been modified and send invalidation messages for a specific Invalidation Group to the Invalidation Manager. It is not enough though: EJB containers are transactional and invalidation messages can only be sent once/if the transaction successfully commits. Consequently, when a bean is modified, an invalidation message is not directly sent to the Manager. Instead, it is kept in a transaction-specific "InvalidationsTxGrouper" instance that will batch-commit all modifications at commit time.

Furthermore, an InvalidationTxGrouper instance can keep track of invalidation messages for more than one Invalidation Group used in the context of the same transaction. At commit time, a **single** invalidation message containing all invalidations for all invalidation groups is sent to the manager that will appropriately dispatch them. This represents a

potentially great optimisation for bridges that will be able to broadcast/publish/send all invalidations in a single message instead of one per invalidation or one per Invalidation Group.

> As the cache invalidation framework will group all invalidation and send them in a single message at the end of the transaction, it is **mandatory** to call the entity beans **inside a transaction**.

EJB Container Configuration

To use cache invalidations in your beans, two configuration steps are required:

1. Container configuration
2. Bean configuration

EJB Container Configuration

During this step you will define a container configuration that includes the invalidator interceptor and the extended cache.

If you are using CMP 2.x, then you are done: an example configuration is already provided as part of `standardjboss.xml`. This container configuration is called "`Standard CMP 2.x EntityBean with cache invalidation`".

If you do not use CMP 2.x or if you want to build your own container configuration, two modifications are necessary:

1. Change the `<instance-cache>` value to `org.jboss.ejb.plugins.InvalidableEntityInstanceCache`

2. Add a new interceptor to the end of the interceptor stack. The class of this interceptor must be `org.jboss.cache.invalidation.triggers.EntityBeanCacheBatchInvalidatorInterceptor`

Here is a sample configuration:

standardjboss.xml

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

   ...
  <container-configuration>
    <container-name>
     Standard CMP 2.x EntityBean with cache invalidation
    </container-name>
    <call-logging>false</call-logging>
    <container-interceptors>
      <interceptor>
       org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.LogInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.SecurityInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.TxInterceptorCMT
      </interceptor>
      <interceptor metricsEnabled="true">
       org.jboss.ejb.plugins.MetricsInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.EntityCreationInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.EntityLockInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.EntityInstanceInterceptor
```

```
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.EntityReentranceInterceptor
      </interceptor>
      <interceptor>
 org.jboss.resource.connectionmanager.CachedConnectionInterceptor
      </interceptor>
      <interceptor>
       org.jboss.ejb.plugins.EntitySynchronizationInterceptor
      </interceptor>
       <interceptor> org.jboss.cache.invalidation.triggers.✂
                     EntityBeanCacheBatchInvalidatorInterceptor
       </interceptor>
       <interceptor>
        org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor
       </interceptor>
     </container-interceptors>
     <instance-
pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
     <instance-cache>
      org.jboss.ejb.plugins.InvalidableEntityInstanceCache
     </instance-cache>
     <persistence-manager>
       org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager
     </persistence-manager>
     <transaction-manager>org.jboss.tm.TxManager
     </transaction-manager>
     <locking-policy>
      org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock
     </locking-policy>
    <container-cache-conf>
      <cache-policy
       >org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-
policy>
      <cache-policy-conf>
        <min-capacity>50</min-capacity>
        <max-capacity>1000000</max-capacity>
        <overager-period>300</overager-period>
        <max-bean-age>600</max-bean-age>
        <resizer-period>400</resizer-period>
        <max-cache-miss-period>60</max-cache-miss-period>
        <min-cache-miss-period>1</min-cache-miss-period>
        <cache-load-factor>0.75</cache-load-factor>
      </cache-policy-conf>
```

```
    </container-cache-conf>
    <container-pool-conf>
      <MaximumSize>100</MaximumSize>
    </container-pool-conf>
    <commit-option>A</commit-option>
  </container-configuration>
...
```

Bean Configuration

For each bean that will fit in the invalidation framework, two things must be defined:

1. The name of container configuration that the bean must use (see previous section)

2. Activate the Invalidation mechanism through the `<cache-invalidation>` tag

jboss.xml

```
  <jboss>
    <enterprise-beans>
        <entity>
         <ejb-name>SimpleEJB</ejb-name>
         <configuration-name
          >Standard CMP 2.x EntityBean with cache invalidation</
         configuration-name>
         <cache-invalidation>True</cache-invalidation>
      </entity>
   </enterprise-beans>
</jboss>
```

Please note that these settings will depends on the Commit Option settings of the container configuration. Thus, the extended cache will only subscribe for invalidation messages if the container runs in Commit Option A or D. Nevertheless, the Invalidator interceptor will

forward invalidation messages independently of the container' Commit Option.

At this point, the container will itself decide what is the name of the **Invalidation Group** by taking the EJB name and will use the default Invalidation Manager instance. If what you are trying to achieve is cluster-wise invalidation of caches, then that is fine because the EJB name will be the same on all nodes, and consequently all EJBs will share the same Invalidation Group name. But if you want to have two identical EJBs running in the same JBoss instance that invalidate their respective cache, they will not belong to the same Invalidation Group because each EJB will have a distinct name. Thus, in this case you will have to assign the same Invalidation Group name to both EJBs. This can be any arbitrary string:

jboss.xml

```
  <jboss>
    <enterprise-beans>
      <entity>
        <ejb-name>SimpleEJB</ejb-name>
        <configuration-name
         >Standard CMP 2.x EntityBean with cache
invalidation</configuration-name>
        <cache-invalidation>True</cache-invalidation>
        <cache-invalidation-config>
          <invalidation-group-name
           >MY_IG_FOR_SIMPLE_EJB</invalidation-group-name>
        </cache-invalidation-config>
      </entity>
      <entity>
        <ejb-name>SimpleEJB-Replica</ejb-name>
        <configuration-name
         >Standard CMP 2.x EntityBean with cache
invalidation</configuration-name>
        <cache-invalidation>True</cache-invalidation>
        <cache-invalidation-config>
          <invalidation-group-name
          >MY_IG_FOR_SIMPLE_EJB</invalidation-group-name>
        </cache-invalidation-config>
```

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

Bridges

Bridges allows to break the single JVM boundary. They can be used so that cache invalidation messages span more than one JBoss instance for example or to enable external actors (database triggers, C programs, etc.) to send invalidation messages to JBoss instances through a specific protocol.

JBoss comes with two bridges implementation out-of-the-box: JMS-based and JBossCluster-based bridges.

JMS-based Bridge

The JMS-based bridge uses a JMS topic to dispatch cache invalidation messages. You can find a default JMS-Bridge configuration commented as part of the “default” configuration:

deploy/cache-invalidation-service.xml

```
...
  <mbean code =
"org.jboss.cache.invalidation.bridges.JMSCacheInvalidationBridgeMBean"
         name="jboss.cache:service=InvalidationBridge,type=JMS">
     <depends>jboss.cache:service=InvalidationManager</depends>
     <depends>
     jboss.mq.destination:service=Topic,name=JMSCacheInvalidationBridge
     </depends>
     <attribute name="InvalidationManager"
      >jboss.cache:service=InvalidationManager</attribute>
     <attribute name="ConnectionFactoryName">
       java:/ConnectionFactory
     </attribute>
     <attribute name="TopicName">
       topic/JMSCacheInvalidationBridge
     </attribute>
     <attribute name="PropagationMode">1</attribute>
```

```
  </mbean>
...
```

The following attributes can be set on this MBean:

- InvalidationManager: The JMX ObjectName of the InvalidationManager to which this bridge must bind
- ConnectionFactoryName and TopicName: name of the factory and topic names used to transmit/receive invalidation messages
- PropagationMode: indicates if the bridge should both send and receive invalidation messages (1), only receive them (2) or only forward them (3).

The JMS bridge is based on a well known service but has mainly two problems:

- As it is not possible for the bridge to know the list of Invalidation Groups that the other JMS bridges have locally, all invalidation messages received by the Invalidation Manager are broadcast, even if other nodes do not use them. This can be a problem for some deployments. Nevertheless, it is still possible to deploy several Invalidation Managers to fragments the Invalidation Groups.
- As all bridges must subscribe to the same JMS Topic, the topic represent a single point of failure
- Due to the asynchronous nature of JSM, the bridge is only able to perform asynchronous invalidations.

JBossCluster-based Bridge

The JBossCluster-based bridge uses the JBoss clustering framework to dispatch cache invalidation messages. A default bridge is automatically deployed as part of the "all" configuration:

deploy/cluster-service.xml

```
...
  <mbean code =
"org.jboss.cache.invalidation.bridges.JGCacheInvalidationBridge"
         name="jboss.cache:service=InvalidationBridge,type=JavaGroups">
     <depends>jboss:service=DefaultPartition</depends>
     <depends>jboss.cache:service=InvalidationManager</depends>
     <attribute name="InvalidationManager">
        jboss.cache:service=InvalidationManager</attribute>
     <attribute name="PartitionName">DefaultPartition</attribute>
     <attribute name="BridgeName">DefaultJGBridge</attribute>
  </mbean>
...
```

The following attributes can be set on this MBean:

- InvalidationManager: name of the Invalidation Manager to which this bridge will bind
- ParitionName: JMX ObjectName of the clustering partition used to exchange invalidation messages
- BridgeName: name of this bridge (optional)

While this bridge requires the clustering to be started, it has several advantages over the JMS-based bridge:

- Each bridge automatically broadcasts (and broadcast updates) of the Invalidation Groups that it has locally. Thus, bridges are able to only forward invalidation messages for Invalidation

Groups that exist on other nodes. Otherwise, the messages will not leave the bridge

- As the bridge uses the clustering infrastructure, there is no single point of failure: any node can die and the invalidation framework will continue to work

- The bridge will use the "synchronous" setting of the Invalidation Manager. Thus, the bridge is able to do fully synchronous distributed cache invalidations.

Use Cases

The cache invalidation framework can be used in two well known architectures that we describe in the next sections.

Single JVM RO/RW bean

In this scenario, a given entity bean (Account) is used in two very different scenarios. First, the web layer must perform a lot of work on it, most exclusively in read-access. At the same time, some other part of the system must be sure to have exclusive write access to the bean for critical account activity.

As both needs are very different, a simple solution is to deploy the same bean twice.

The first deployment, AccountRO (account read-only) uses commit-option A (or D if some other application can directly write to the DB) and has the <read-only> tag set to true. Thus, the web layer can work very efficiently with the accounts because no locking will occur and every information can be read from cache.

The second deployment, AcountRW (account read-write), uses commit-option C (or even A if no other application writes to the DB) and uses pessimistic locking at the JBoss level. Thus, the critical part of the system can be sure to use up-to-date information and have exclusive access to it.

The problem now is that as AccountRW is used to modify data, the AccountRO cache will no more be in-synch. In this case, simply complete the EJB deployment descriptor so that both beans are linked and share cache invalidation messages:

jboss.xml

```
  <jboss>
    <enterprise-beans>
      <entity>
        <ejb-name>AccountRW</ejb-name>
        <configuration-name
         >Standard CMP 2.x EntityBean with CI in Commit Option C<
        /configuration-name>
        <cache-invalidation>True</cache-invalidation>
        <cache-invalidation-config>
          <invalidation-group-name>AccountCacheGroup</
           invalidation-group-name>
        </cache-invalidation-config>
      </entity>
      <entity>
        <ejb-name>AccountRO</ejb-name>
        <configuration-name
         >Standard CMP 2.x EntityBean with cache
invalidation</configuration-name>
        <cache-invalidation>True</cache-invalidation>
        <cache-invalidation-config>
          <invalidation-group-name> AccountCacheGroup</
           invalidation-group-name>
        </cache-invalidation-config>
      </entity>
   </enterprise-beans>
 </jboss>
```

Thus, each time AccountRW is used to modify an entity, the associated entry in AccountRO's cache will be automatically invalidated.

RO/RW cluster

In this scenario, a given entity bean (StockQuote) is not frequently changed but heavily accessed by web clients in read-only mode (simple display of stock quotes). As the web load is big, a cluster of JBoss instances is used. In order not to flood the database with identical requests, we want to be able to use the EJB cache.

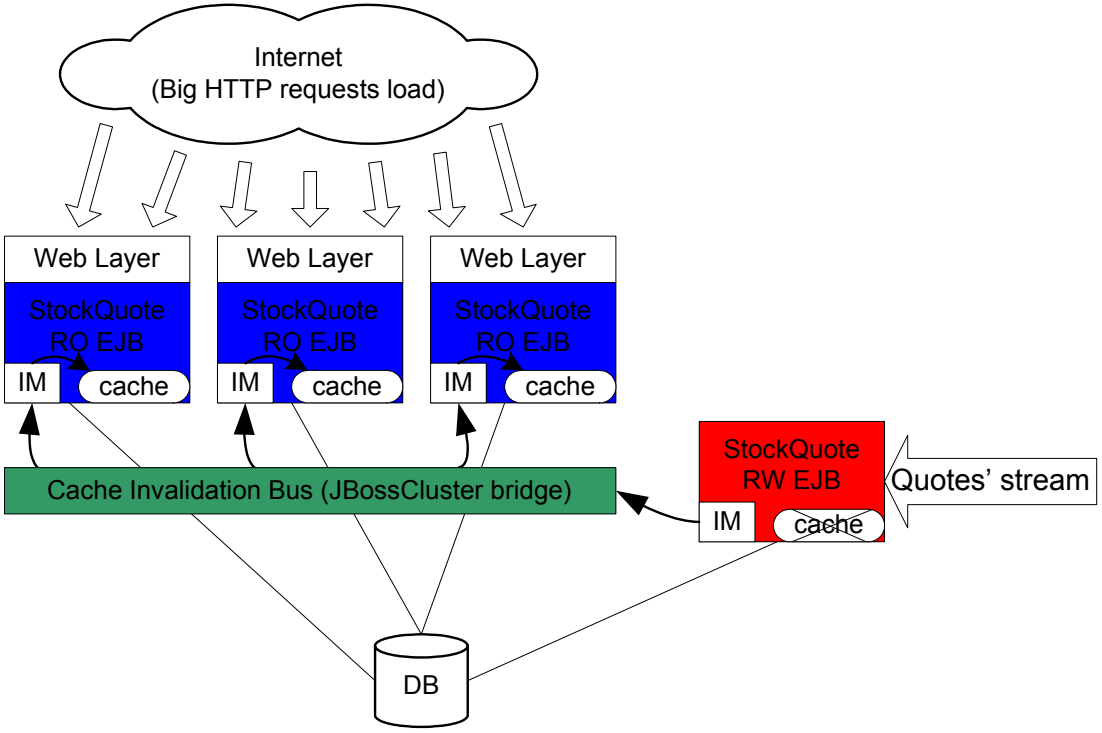


Figure 12RO/RW Cluster Using Cache Invalidations

In this case all read-only nodes will deploy an optimised StockQuote EJB configuration using Commit Option A and the <read-only> tag to prevent pessimistic locking performance degradation.

The read-write node(s) will use a container configuration using Commit Option C and pessimistic locking.

With such a configuration, the cluster is able to server high request loads without impacting the database. Without the cache invalidation framework, Commit Option B/C should have been used (or D in JBoss if serving non totally up-to-date data is not critical) which would have had a big impact on the database load: each HTTP query would generate at least one database query.

Chapter 9

9. Clustering Architecture

Detailed review of JBoss 3.0 clustering architecture

This section will introduce the main classes that form the clustering framework in JBoss. They can be used to develop your own clustered services.

Overview

Clustering in JBoss is based on a simple framework of tools that are used to provide all other services.

This framework is itself based on a communication framework that is completely abstracted. Consequently, potentially any communication framework providing a strong enough semantic can be plugged in. The current implementation uses JGroups (**http://www.jgroups.org**).

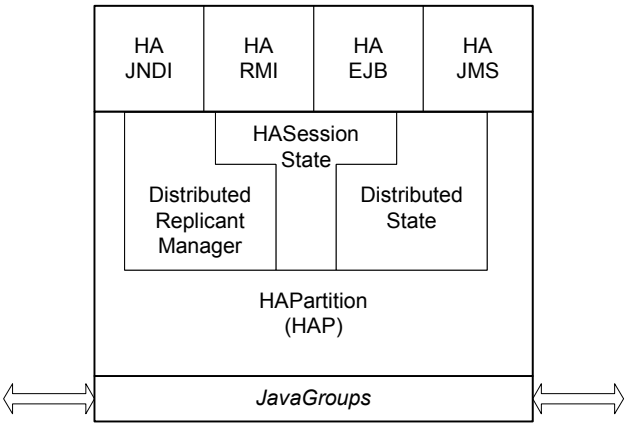


Figure 13. JBoss clustering building blocks

JBoss Clustering Framework

HAPartition

The HAPartition abstract the communication framework and provide access to a basic set of communication primitives. You do not directly built an instance of a HAPartition. Instead you use an MBEAN definition (or equivalent).

First, the HAPartition provides informational data: the name of the cluster and the name of this node (dynamically built at connection time).

```
public String getNodeName();
public String getPartitionName();
```

During the lifetime of the partition, nodes will join and leave the partition. To track this, different methods are available.

Each time a node leave or join, the cluster topology changes: we call this a view, i.e. a list of member nodes. Furthermore, each view has an unique identifier.

It is also possible to register to receive a callback each time the membership of the cluster has changed.

```
// View and view ID
//
public long getCurrentViewId();
public Vector getCurrentView ();

// Membership callbacks
//
public interface HAMembershipListener
{
public void membershipChanged(Vector deadMembers, Vector newMembers,
Vector allMembers);
}
public void registerMembershipListener(HAMembershipListener
listener);
public void unregisterMembershipListener(HAMembershipListener
listener);
```

Now on the communication aspects, two categories of primitives take place in HAPartition: state transfer and RPC calls.

Different services will use the services of a HAPartition. Consequently, each of these services will first need to register in the HAPartition with a specific key name:

> Services around the network with the same key name that have subscribed in a HAPartition with the same name, are considered as being services replicas.

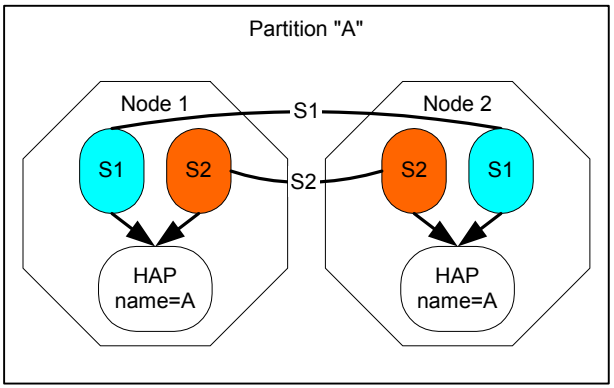


Figure 14. Replicas in a cluster

In the figure above, two nodes belong two the same partition (named "A"). On each node, two services (S1 and S2) have subscribed to the HAPartition with their corresponding name ("S1" and "S2"). Consequently, S1 and S2 are now clustered and any RPC call they would made would only be received by their counterparts that have subscribed with the same service name. Thus, the HAPartition acts as a service multiplexer.

```
// (un-)register a new service with this HAPartition
//
public void registerRPCHandler(String objectName, Object handler);
public void unregisterRPCHandler(String objectName, Object
subscriber);

// Called only on all members of this partition on all nodes
//
public ArrayList callMethodOnCluster(String objectName,
                                     String methodName,
                                     Object[] args,
                                     boolean excludeSelf)
   throws Exception;
public void callAsynchMethodOnCluster (String objName,
                                     String methodName,
                                     Object[] args,
                                     boolean excludeSelf)
```

```
    throws Exception;
```

Once registered (through registerRPCHandler), a service can receive RPC calls from other nodes. RPC calls are made through the callMethodOnCluster and callAsynchMethodOnCluster. The first method returns, in an array, the serialized answers of all other nodes while the second method is an asynchronous method call that returns directly (and thus provides no answers).

The parameters meaning is as follow:

- `objName`: the name of the target service.
- `methodName`: the name of the remote Java Method to be called
- `args`: the parameters of the method to be called
- `excludeSelf`: a boolean indicating if the call must also be made on the current node.

The following code snipset shows how a service will call a method on remote services:

Listing 9-1. Example of clustered code

```
public class WordPrinter
{
public WordPrinter (HAPartition myPartition)
{
     myPartition.registerRPCHandler (“WordPrinter”, this);
}

public void printWordOnOtherNodes (String theWord)
{
     Object[] args = {theWord};
     callAsynchMethodOnCluster (“WordPrinter”, “_printWord”, args,
                                true);
}
```

```
public void _printWord (String theWord)
{
     System.out.println (“WordPrinter : ” + theWord);
}
}
```

The second category of primitive concerns state transfer. In clustered systems, state transfers has an important place. If we describe a node as a system having a set of primitives and a state at time T1, then, if a new node joins the cluster, it needs to initialise and get the current state to be able to work coherently with the rest of the cluster.

While some service may be completely stateless, others maintain a clustered state and when a new node joins, need to inform the same service on the new node of the current state. Nevertheless, a state transfer operation should not be seen as a standard clustered RPC call: we need to be certain that the state given by an already running node to the new node fits perfectly in the sequence of messages (RPC calls, etc.) send to the whole cluster. Because any message may potentially change the state, the position of the state transfer in the flow of messages is highly important. In the following sequence, for example, the state transfer is not correct:

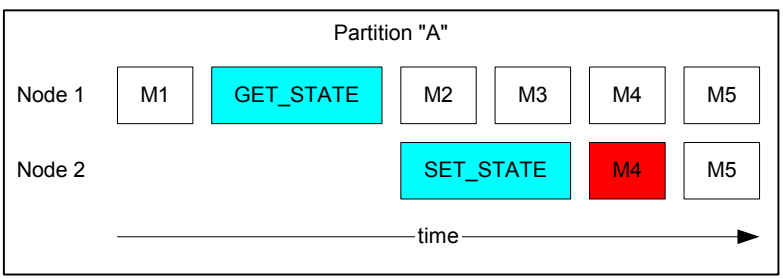


Figure 15. State transfer process

The state that Node 1 gives is relative to message M1 (that may have changed the state) and to the fact that messages M2-5 have not yet been able to modify the state. But the state that Node 2, the joining

node, receives is followed by message M4. Consequently, Node 2 misses messages M2 and M3: this is a problem because we don't know what could have been their effect on the state.

To subscribe to state transfer, the following part of the interface of HAPartition has to be used:

```
public interface HAPartitionStateTransfer
{
     public Serializable getCurrentState ();
     public void setCurrentState(Serializable newState);
}

public void subscribeToStateTransferEvents (String objectName,
     HAPartitionStateTransfer subscriber);
public void unsubscribeFromStateTransferEvents (String objectName,
     HAPartitionStateTransfer subscriber);
```

The following sections introduce the Distributed Replicant Manager and Distributed State services. As they belong to the clustering framework, access methods are provided from the HAPartition to these services. This may change in the future.

```
public DistributedReplicantManager getDistributedReplicantManager();
public DistributedState getDistributedStateService ();
```

Distributed Replicant Manager (DRM)

The Distributed Replicant Manager is a very useful service used in many parts of the clustering. Its role is to manage, for a given key, a list of serialised data each owned by a specific node. Behind this strange definition is something rather simple.

Imagine you want to manage a list of stubs for a given RMI server running on different nodes. Each node has this RMI server, hence a stub to share with other nodes. The Distributed Replicant Manager allows to share these stubs in the cluster and know to which node a stub belongs. Furthermore, one useful feature is that if one node

crashes, its entry is automatically removed from the list of replicated data (and services can register to receive a callback when it happens).

Furthermore, for each key, an id is available (see getReplicantsViewId). This id, for a given set of replicants, is identical on all nodes of the cluster. Each time the set of replicants changes, this id also changes. This allows, for example, a client to determine if its view of a particular set of replicants is up to date.

To manage the replicants, the interface propose a set of methods:

```
/**
 * Add a replicant, it will be attached to this cluster node
 */
public void add(String key, Serializable replicant) throws Exception;

/**
 * remove the entire key from the ReplicationService
 */
public void remove(String key) throws Exception;;

/**
 * Lookup the replicant attached to this cluster node
 */
public Serializable lookupLocalReplicant(String key) throws
Exception;

/**
 * Return a list of all replicants.
 */
public List lookupReplicants(String key) throws Exception;

/**
 * Returns an id corresponding to the current view of this set of
replicants.
*/
public int getReplicantsViewId(String key);
```

Furthermore, a subscription mechanism allows to receive a callback when the number of replicants for a specific key changes:

```
/**
 * When a particular key in the DistributedReplicantManager table
gets modified, all listeners
 * will be notified of replicant changes for that key.
 */
public interface ReplicantListener
{
   public void replicantsChanged(String key,
     List newReplicants ,
     int newReplicantsViewId);
}

public void registerListener(String key,
                             ReplicantListener subscriber)
   throws Exception;
public void unregisterListener(String key,
                               ReplicantListener subscriber)
   throws Exception;
```

In a subsequent section, we will introduce the HARMIServer class that is based on this service.

On last interesting feature is available through the last method:

```
public boolean isMasterReplica (String key);
```

In your application, if you need to elect a master node from the set of nodes replicating a given key, you can use this helper method. It will elect only one of the nodes of the cluster that replicate the given key as being the master node. Thus, although there is no master-slave notion needed in the DRM service, the service can help you elect one of the *replicant* nodes.

Distributed State (DS)

The Distributed State Service allows to share cluster-wide a set of dictionary. This service can be used, for example, to store settings or parameters that should be used by all containers in the cluster.

The service first proposes methods to add, remove and consult these dictionaries (each dictionary is identified by a name called a category):

```
/**
 * Associates a value to a key in a specific category
 */
public void set(String category, String key, Serializable value)
   throws Exception;

/**
 * remove the entire key from the ReplicationService
 */
public void remove(String category, String key)
   throws Exception;

/**
 * Lookup the replicant attached to this cluster node
 */
public Serializable get(String category, String key)
   throws RemoteException;

/**
 * Return a list of all categories.
 */
public Collection getAllCategories()
  throws RemoteException;

/**
 * Return a list of all keys in a category.
 */
public Collection getAllKeys(String category)
   throws RemoteException;

/**
 * Return a list of all values in a category.
 */
public Collection getAllValues(String category)
   throws RemoteException;
/**
 * The service also proposes a subscription mechanism to be informed
 * when a particular dictionary changes:
 * When a particular key in the DistributedState table gets modified,
 * all listeners
```

```
 * will be notified of replicant changes for that key. Keys are
 * organized in categories.
 */
public interface DSListener
{
     public void valueHasChanged(String category, String key,
                                 Serializable value);
     public void keyHasBeenRemoved (String category, String key,
                                    Serializable previousContent);
}

public void registerDSListener(String category,
                               DSListener subscriber)
     throws RemoteException;
public void unregisterDSListener(String category,
                                 DSListener subscriber)
     throws RemoteException;
```

HA-RMI

Through the HAPartition framework, it is possible to provide load-balancing and fail-over facilities for your RMI Servers.

First, you need to prepare the HARMIServer. You need to tell it which HAPartition to use as its basis for clustering communication, the name of the current RMI server (i.e. an string identifier distinct from any other HARMIServer running on the same node) and the target object. Note that the target object no longer needs to be exported through RMI! You can also specify if a specific port, client and server socket factory and IP binding should be used. If that is not the case you can use default values (0 for the port and null for the other attributes).

```
…
HAPartition myPartition = (HAPartition)ctx.lookup("/HAPartition/" +
partitionName);
HARMIServer rmiserver = new HARMIServerImpl(myPartition, "MyService",
     MyService.class, myService, 0, null, null, null);
MyService stub = (MyService)rmiserver.createHAStub(new RoundRobin());
…
```

The obtained stub can now be obtained (in return from a RMI call or through a JNDI lookup) and used by any remote client!

But what if, as mentioned above, you already have your own smart stub implemented?

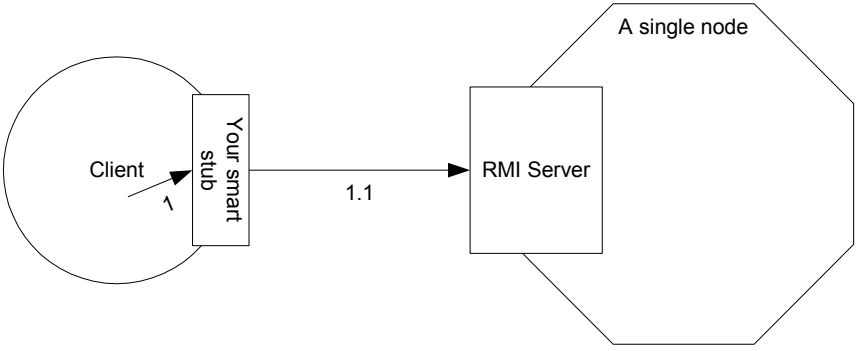


Figure 16. Standard smart-stub to RMI server

Easy. Instead of using the reference of your Remote server in your smart stub, use the stub returned by the two lines of code above: we will then have two levels of stubs!

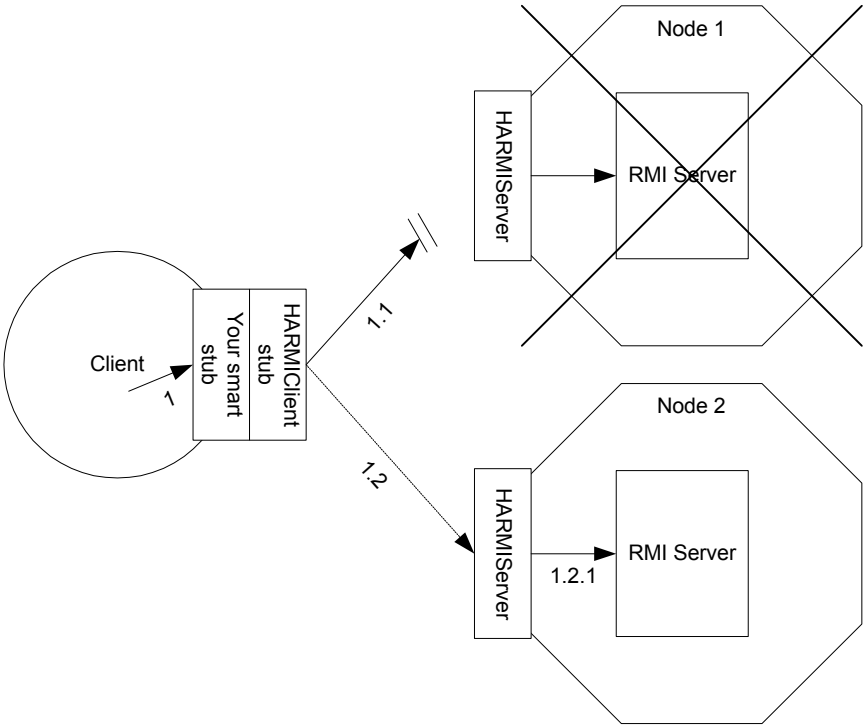


Figure 17. HARMIServer coupled with a pre-existing smart stub

Updating JNDI

There is a potential problem though with the solution described above: upon registering the proxy with JNDI, it is serialized. Thus, when nodes are started/stopped, while the proxy object is refreshed with the new list of target servers, the serialized representation stored in JNDI is not. Consequently, clients may get an outdated version of the proxy from JNDI and this one will never be refreshed.

To avoid this, you can request the HARMIServerImpl object to call you back upon changes on the proxy object you have generated:

```
HAPartition myPartition = (HAPartition)ctx.lookup("/HAPartition/" +
partitionName);
```

```
HARMIServer rmiserver = new HARMIServerImpl(myPartition, "MyService",
     MyService.class, myService, 0, null, null, null, this);
MyService stub = (MyService)rmiserver.createHAStub(new RoundRobin());
…
```

The last parameter you pass to the HARMIServerImpl constructor is a reference to a callback object that implements the HARMIProxyCallback interface:

```
public interface HARMIProxyCallback
{
    public void proxyUpdated ();
}
```

The HARMIServerImpl will call this method on the callback object you have provided each time its list of target servers is updated. Here is an possible implementation:

```
import org.jboss.ha.framework.interfaces.HARMIProxyCallback;
import org.jboss.ha.framework.interfaces.HAPartition;
import org.jboss.ha.framework.interfaces.RoundRobin;
import org.jboss.ha.framework.server.HARMIServerImpl;

import javax.naming.InitialContext;

public class TestHARMIServer
   extends org.jboss.system.ServiceMBeanSupport
   implements TestHARMIServerMBean, HARMIProxyCallback
{
   private String jndiName = "MyJndiName";
   private String partitionName = "DefaultPartition";
   HAPartition partition = null;

   private MyServiceImpl myServiceImpl;
   private HARMIServerImpl myHARMIServer;

   protected Object stub = null;

   public String getJndiName()
```

```
    {
       return jndiName;
    }

    public String getPartitionName()
    {
       return partitionName;
    }

    public void startService() throws Exception
    {
       // get access to the HAPartition (i.e. the cluster)
       String fullPartitionName = "/HAPartition/" + partitionName;
       InitialContext context = new InitialContext();
       partition = (HAPartition) context.lookup(fullPartitionName);

       // Create a clustered Proxy/stub
       myServiceImpl = new MyServiceImpl(partition);
       myHARMIServer = new HARMIServerImpl(partition, "MyServiceId",
                       myServiceInterface.class, myServiceImpl,
                       0, null, null, null, this);
       stub = myHARMIServer.createHAStub(new RoundRobin());

       rebind();
    }

    public void stopService()
    {
       this.stub = null;
       unbind(jndiName);
    }

    private void rebind() throws Exception
    {
       log.info("Rebinding...");
       context.rebind(jndiName, stub);
    }

    private void unbind(String jndiName) throws Exception
    {
       log.info("Unbinding...");

       new InitialContext().unbind(jndiName);
       myHARMIServer.destroy();
```

```
   }

   // This IS the callback we receive when the list of targets
   // in the proxy is updated => we simply rebind our proxy in JNDI
   // which will re-serialize its updated content.
   public void proxyUpdated()
   {
      try
      {
         rebind();
      }
      catch (Exception ignored) {}
   }
}
```

Chapter 10

10. Clustering Your Own Services

Use JBoss clustering services to provide clustered behavior for your applications

To cluster your services, only your imagination is the limit! You can use the clustering framework services as well as any other helper class (such as HARMIServer for example) or HA-JNDI to reach your goal.

Nevertheless, try not to forget the following guidelines:

- If you have developed clustered services (MBeans, etc.) that are very stable, you should most probably install a new HAPartition for your own usage and leave the default one for JBoss clustering features.
- When receiving callbacks from the clustering framework, take care to reduce as much as possible the time spent in the callback method: your code needs to be co-operative with other services using the same underlying HAPartition.
- State transfer is a costly operation: try to reduce the amount of data you need to transfer between nodes to the minimum.

Chapter 11

11. Other Clustering Services

Other services provided as part of JBoss that can be used as part of clustering scenarios.

Singleton Service

Provided by Ivelin Ivanov

A clustered singleton is a service that is deployed on multiple nodes in a cluster, but is running on only one of the nodes. The node running the singleton service is typically called master node. When the master fails, another master is selected from the remaining nodes and the service is restarted on the new master.

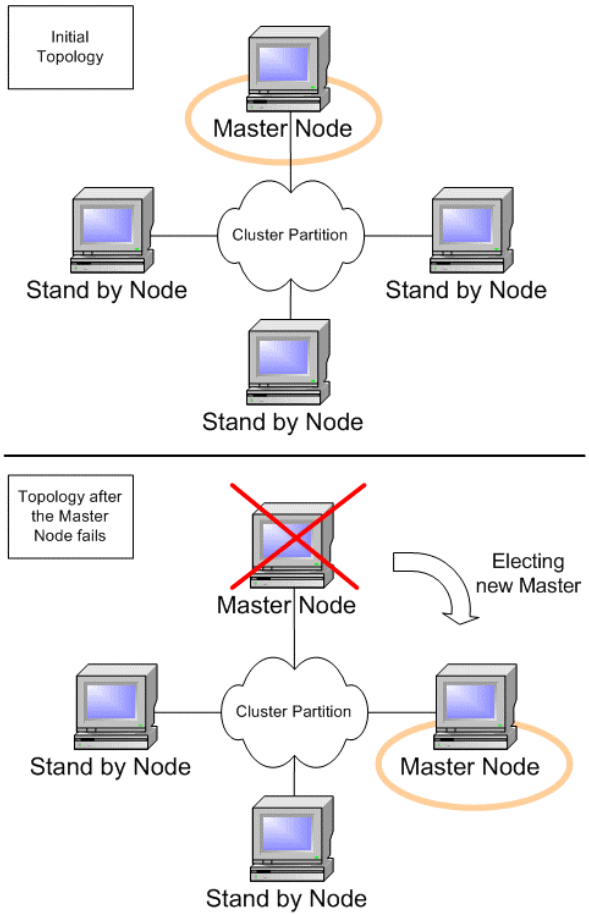


Figure 18. Clustered Singleton Service

In real world applications, there are often times when some task needs to be executed upon startup or as a result of an application event. While it is fairly easy to implement such a task in a single JVM, the solution will usually not work immediately in a clustered environment. Even in the simple case of a task activated upon startup on one of the nodes in a two node cluster, there are several problems that need to be addressed:

- when the application is started simultaneously on both nodes, which VM should run the startup task
- when the application is started on the first node and much later on the second, how would the second node know not to run the startup task again
- when the node that started the task fails, how would the other one know to resume the task
- when the node that started the task fails, but later on comes back up, how to ensure that the task remains running on only one of the nodes

The logic to solve these problems is unlikely to be included in the design of a single JVM solution. However a solution can be found to address the case at hand and it can be patched on the startup task. This is an acceptable approach for a few startup tasks and two node clusters.

Going forward, as the application grows and becomes more successful, there might be a need for other startup tasks. It may also need to scale to more than two nodes. The clustered singleton problem can quickly become mind boggling for larger clusters, where the different node startup scenarios are far not as easy to enumerate as in the two node case. Another factor which complicates the problem even more is the communication efficiency. While two nodes can directly connect to each other and negotiate, 10 nodes will have to establish 45 total connections if they want to use the same technique.

This is where JBoss comes handy. It eliminates most of the complexity and allows application developers to focus on building singleton services invariant of the cluster topology.

We will illustrate how the JBoss clustered singleton facility works with an example.

First, we will need a service archive descriptor. Let's use the one that ships with JBoss under server/all/farm/cluster-examples-service.xml. Following is an excerpt from it:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
   <!--
    | This MBean is an example of a cluster Singleton
    -->
   <mbean
     code="org.jboss.ha.singleton.examples.HASingletonMBeanExample"
     name="jboss:service=HASingletonMBeanExample">
   </mbean>

   <!--
    | This is a singleton controller which works similarly to the
    | SchedulerProvider (when a MBean target is used)
    -->
   <mbean code="org.jboss.ha.singleton.HASingletonController"
          name="jboss:service=HASingletonMBeanExample-✂
                HASingletonController">
      <depends>jboss:service=DefaultPartition</depends>
      <depends>jboss:service=HASingletonMBeanExample</depends>
      <attribute name="TargetName"
       >jboss:service=HASingletonMBeanExample</attribute>
      <attribute name="TargetStartMethod">startSingleton</attribute>
      <attribute name="TargetStopMethod">stopSingleton</attribute>
   </mbean>
</service>
```

This file declares two MBeans: HASingletonMBeanExample and HASingletonController. The first one is an example of a singleton service which contains the custom code. It is a simple JavaBean with the following source code:

```
public class HASingletonMBeanExample
  implements HASingletonMBeanExampleMBean
{

  private boolean isMasterNode = false;

  public void startSingleton()
  {
    isMasterNode = true;
  }

  public boolean isMasterNode()
  {
    return isMasterNode;
  }

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

All the custom logic for this particular singleton service is contained within this class. Our example is not very useful. It simply indicates, via the isMasterNode member variable, whether it is the master node running the singleton. The value of isMasterNode will be true on only one of the nodes in the cluster where it is deployed. HASingletonMBeanExampleMBean exposes this variable as an MBean attribute. It also exposes startSingleton() and stopSingleton() as managed MBean operations. These methods control the lifecycle of the singleton service. They are invoked by JBoss automatically when a new master node is elected.

How does JBoss control the singleton lifecycle throughout the cluster? The answer to this question is in the MBean declarations. Notice that the HASingletonMBeanExample-HASingletonController MBean is given the name of the sample singleton MBean and its start and stop methods. On each node in the cluster where these MBeans are

deployed, the controller will work with all the other controllers with the same MBean name deployed in the same cluster partition and oversee the lifecycle of the singleton. The controllers are responsible for keeping track of the cluster topology. Their job is to elect the master node of the singleton upon startup as well as to elect a new master should the current one fail or shut down. In the latter case, when the master node shuts down gracefully, the controllers will wait for the singleton to stop, before starting another instance on the new master node.

A singleton service is scoped in a certain cluster partition via its controller. Notice that in the declaration above the controller MBean is dependent on the MBean service DefaultPartition. If the partition where the singleton should run is different than the defaul, its name can be provided to the controller via the MBean attribute - PartitionName.

Clustered singletons are usually deployed via the JBoss farming service. To test this example, just drop the service file above in the server/all/farm directory and go to the JBoss JMX web console. You should be able to see the following:

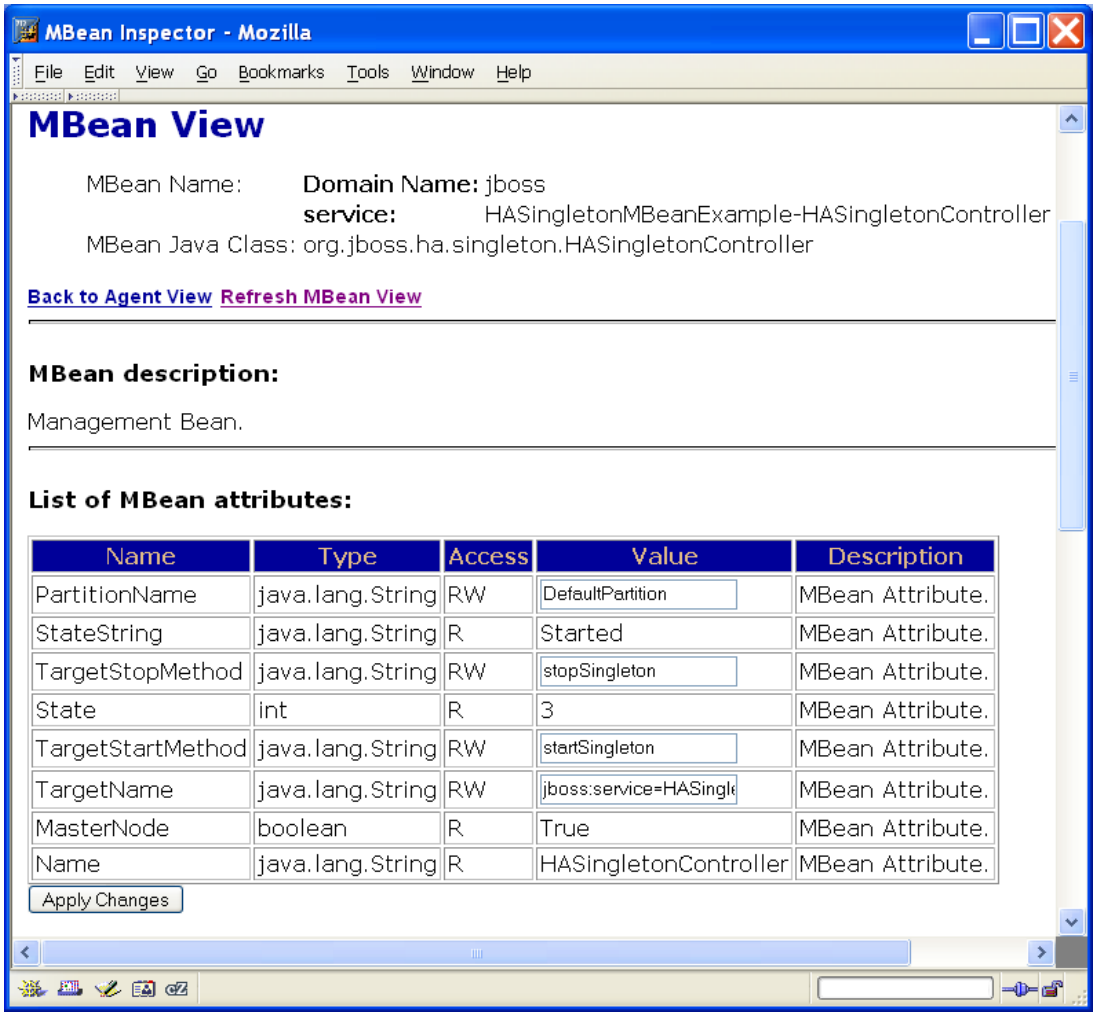


Figure 19. Controller MBean View. “MasterNode” is True on only one of the nodes

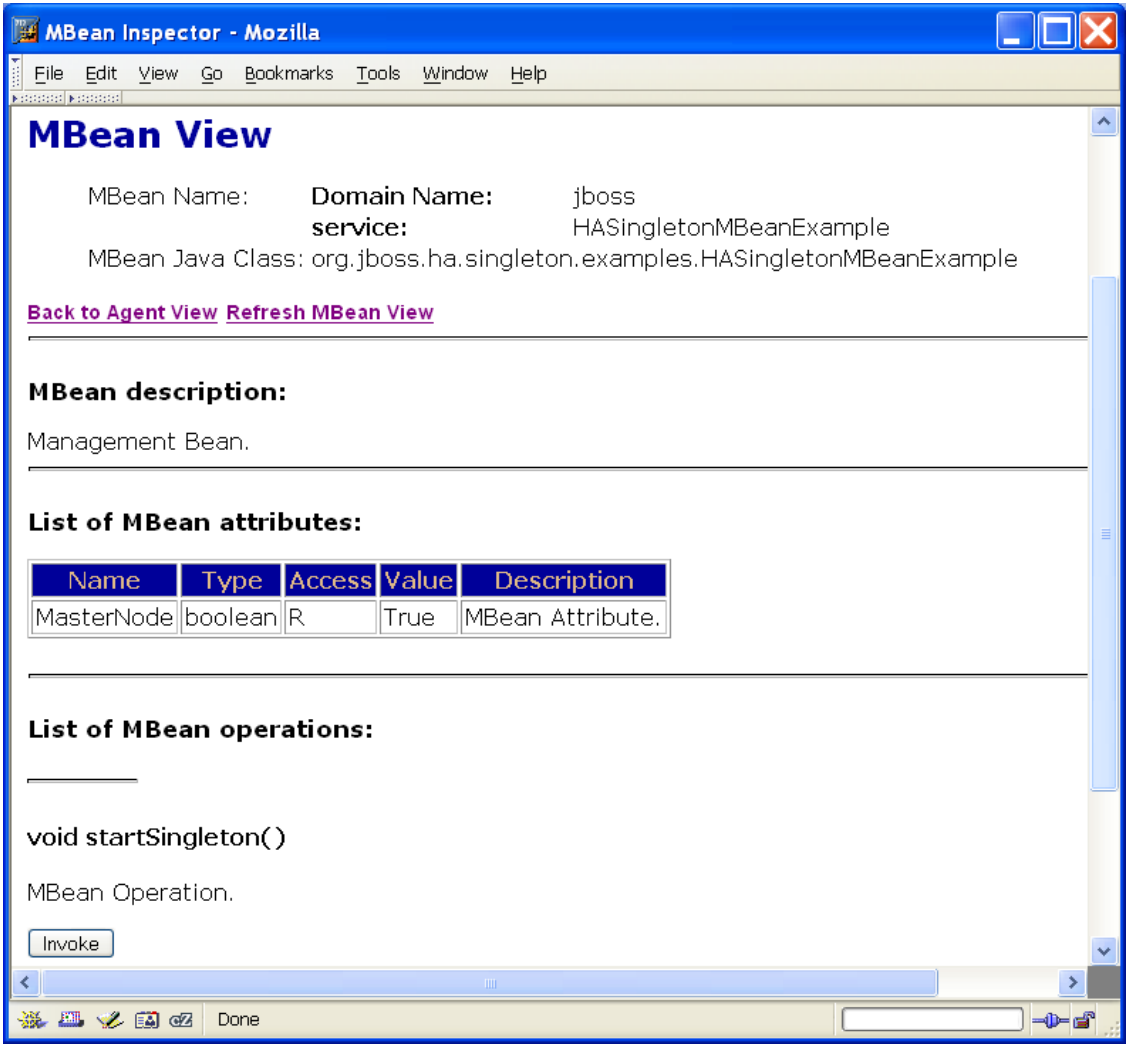


Figure 20. Sample singleton MBean View. "MasterNode" has the same value as "MasterNode" on the controller MBean

Scheduler Service

Provided by Ivelin Ivanov

The JBoss 3.x Scheduler service is covered in detail by the JBoss 3.x - Administration and Development book. It has three interdependent components:

- ScheduleManager, which serves as a centralized registry for registering and executing schedules.
- ScheduleProvider, which abstracts the custom logic that creates schedules
- Schedulable, which represents the abstraction of a task that is executed at scheduled times

Often, applications need to schedule tasks which have to be executed once in the scope of the application, independent of whether it is running stand alone or in a cluster of multiple nodes. Examples for such tasks include regular database cleanup, email notifications and scheduled reports.

All of the JBoss ScheduleProvider services accept an MBean attribute which enables them to schedule tasks on only one node in the cluster (of one or more nodes). The attribute name is HASingleton and its value is a boolean type. When set to True, all Schedule provider MBeans registered with the same name and deployed in the same cluster partition, will coordinate and make sure that the schedule is only provided to the schedule manager on one of the nodes. When set to false, each of the schedule providers will act independently and as a result they will all schedule tasks with their local scheduler managers. The default value of the attribute is true, which allows transparent transition of stand-alone schedule providers to a clustered environment.

The name of the partition where a singleton schedule provider service will be deployed can be set via the PartitionName attribute. By default the value assumes the default JBoss partition name.

Here is an excerpt from the example service archive descriptor cluster-examples-service.xml, located in the server/all/farm directory in the JBoss installation.

```
    <!--
     | This MBean is an example of an HA Schedule Target
     | which is identical to a regular Schedule Target
     | (the example class is the same, just the MBean has
     | different names)
     - ->
    <mbean
      code="org.jboss.varia.scheduler.example.SchedulableMBeanExample"
      name="jboss:service=HASchedulableMBeanExample">
    </mbean>
    <!- - -->


    <!--
     | The Schedule Manager has to be started whenever
     | schedules are needed.
     |
     | Uncomment only if not started by
     | another service (e.g. schedule-manager-service.xml)
     - ->
    <mbean code="org.jboss.varia.scheduler.ScheduleManager"
           name="jboss:service=ScheduleManager">
       <attribute name="StartAtStartup">true</attribute>
    </mbean>
    <!- - -->

    <!--
     | This is a single schedule Provider which works like the
     | one in schedule-manager-service.xml
     |
     | The key difference is the explicit use of the
```

```
    | HASingleton MBean attribute
    | to make the provider a clustered singleton.
    | When HASingleton is set to true the MBean will usually declare
    | dependency on a cluster partition. In this case it is the
    | DefaultPartition.
    | When not explicitly set the attribute defaults to true.
    |
    | The same attribute can also be used for the other schedule
providers as well:
    | DBScheduleProvider and XMLScheduleProvider
    |
    |
    - ->
   <mbean code="org.jboss.varia.scheduler.SingleScheduleProvider"
          name="jboss:service=HASingleScheduleProvider">
      <depends>jboss:service=DefaultPartition</depends>
      <depends>jboss:service=ScheduleManager</depends>
      <depends>jboss:service=HASchedulableMBeanExample</depends>
      <attribute name="HASingleton">true</attribute>
      <attribute name="ScheduleManagerName"
       >jboss:service=ScheduleManager</attribute>
      <attribute name="TargetName"
       >jboss:service=HASchedulableMBeanExample</attribute>
      <attribute name="TargetMethod"
       >hit( NOTIFICATION, DATE, REPETITIONS, SCHEDULER_NAME,✂
             java.lang.String )</attribute>
      <attribute name="DateFormat"></attribute>
      <attribute name="StartDate">NOW</attribute>
      <attribute name="Period">10000</attribute>
      <attribute name="Repetitions">10</attribute>
   </mbean>
   <!- - -->
```

The deployment descriptor above is almost identical to the one provided by schedule-manager-service.xml, but is intended to be deployed via farming. Although not necessary, since *true* is the default value, the descriptor specifies the HASingleton attribute of the schedule provider for illustration of its usage.

Notification Service

Provided by Ivelin Ivanov

When it comes to reliable, mission critical and sophisticated J2EE messaging, there is one API that stands out - JMS. The JBossMQ service is a robust JMS implementation which meets the high standards commanded by the API for distributed, transactional and secure messaging. Then why would we need another notification service?

The most typical clustered applications are deployed in a symmetric, homogeneous environment where components need to notify each other about various life cycle or domain change events, much like AWT and Swing widgets exchange events. This is where the JBoss cluster notifications come in place. They fill the need for reliable, quick and lightweight events. Implemented in compliance with the JMX notifications API, the cluster notifications do not require new skills from developers familiar with JMX.

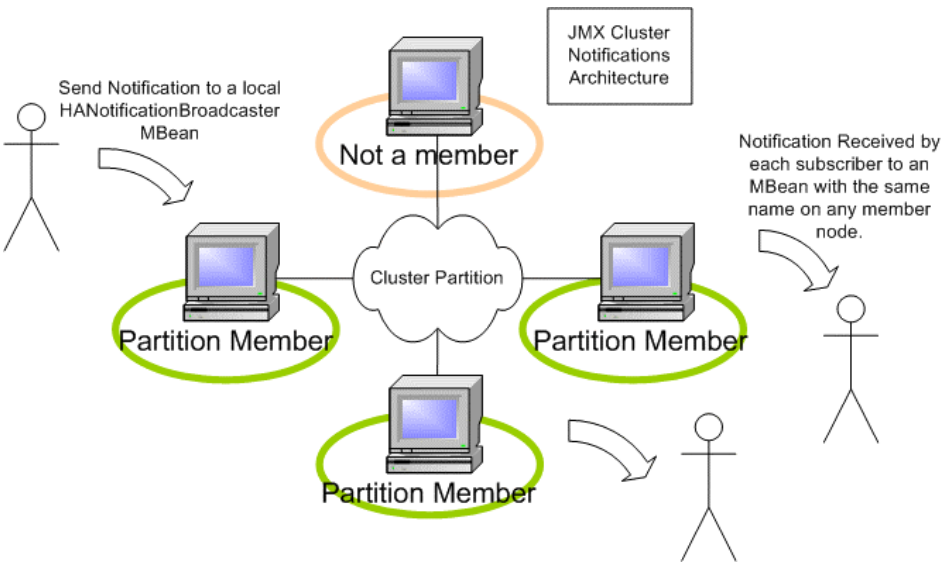


Figure 21. Clustered Notification Service

If you have used JMX notifications before, then the clustering extension should be a breeze. There are two common ways to employ clustered notifications: by extending HAServiceMBeanSupport or delegating the work to instances of this class.

Let's look at an example which comes with the JBoss distribution.

Here is an excerpt from the familiar descriptor cluster-examples-service.xml, located in the server/all/farm.

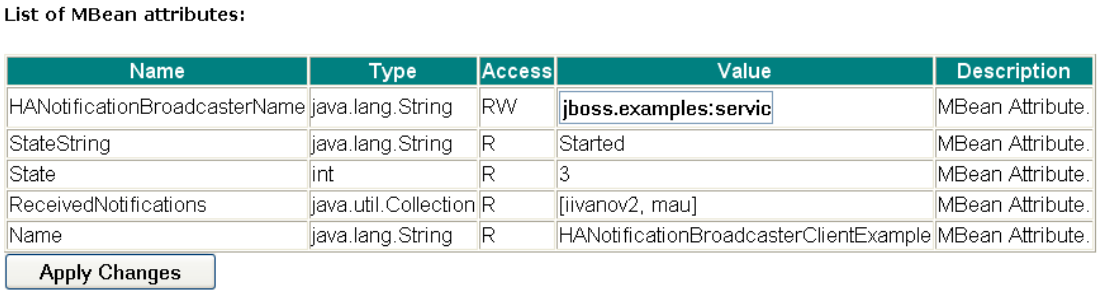
```
<!--
    | This MBean is an example showing how to extend a cluster
notification
    | broadcaster
    | Use the sendNotification() operation to trigger new clustered
notifications.
    | Observe the status of each instance of this mbean in the
participating
    | cluster partition nodes.
    - ->
   <mbean
    code="org.jboss.ha.jmx.examples.HANotificationBroadcasterExample"
    name="jboss.examples:service=HANotificationBroadcasterExample">
      <depends>jboss:service=DefaultPartition</depends>
   </mbean>
   <!- - -->

   <!--
    | This MBean is an example that shows how to delegate
    | notification services to a HANotificationBroadcaster.
    | Use the sendNotification() operation to trigger new clustered
    | notifications.
    | Observe the status of each instance of this mbean in the
    | participating cluster partition nodes.
    | ->
   <mbean code="org.jboss.ha.jmx.examples.✂
```

```
                HANotificationBroadcasterClientExample"
name="jboss.examples:service=HANotificationBroadcasterClientExample">
     <depends>jboss.examples:service=HANotificationBroadcasterExample
</depends>
     <attribute name="HANotificationBroadcasterName"
>jboss.examples:service=HANotificationBroadcasterExample</attribute>
   </mbean>
```

The MBean implemented by the HANotificationBroadcasterExample class provides common clustering services like shared distributed state, replication management and notifications. It is a convenience class that can be extended and customized or can be used as is. Its most important attribute is the PartitionName, which fixates the partition of nodes where all MBeans with the same name will work together.

The other MBean in this example is implemented by the HANotificationBroadcasterClientExample class. It uses the former MBean's services to broadcast notifications to the local listeners as well remote cluster listeners that subscribed to the HANotificationBroadcasterExample MBean. At the same time the client MBean will also receive all notifications sent through any instance of the broadcaster. Clear as mud?

Maybe a picture will help. Here is a screenshot of the client MBean attributes:



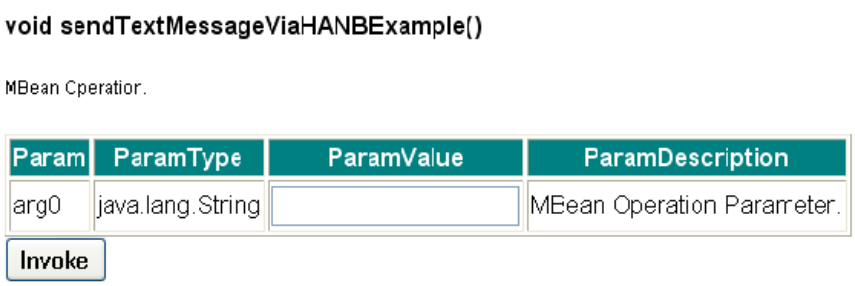
List of MBean attributes:

Name	Type	Access	Value	Description
HANotificationBroadcasterName	java.lang.String	RW	jboss.examples:servic	MBean Attribute.
StateString	java.lang.String	R	Started	MBean Attribute.
State	int	R	3	MBean Attribute.
ReceivedNotifications	java.util.Collection	R	[iivanov2, mau]	MBean Attribute.
Name	java.lang.String	R	HANotificationBroadcasterClientExample	MBean Attribute.

Apply Changes

The first attributes points to the name of the cluster notification broadcaster we will subscribe to. The next attribute of interest is ReceivedNotifications. When a message is sent to any of a deployed instance of the broadcaster MBean on any of the partition nodes, it will be received and listed in the Value column for this attribute. In the screenshot above, there are 2 received notifications - "iivanov2" and "mau". Even though it is not obvious from the picture, one of these messages was received from a remote host.

Now, a screenshot of the MBean operation of interest:



The name is not very friendly, but it is distinct and will do for this example. When this operation is invoked with a text message as argument, the value will appear in the ReceivedNotification attribute on each of the client MBeans deployed in the same partition.

Let's look at the code of the client MBean to see that it is actually straightforward. I made an attempt to annotated the code and keep it simple, so that there is no need for additional analysis.

```
public class HANotificationBroadcasterClientExample
  extends ServiceMBeanSupport
  implements HANotificationBroadcasterClientExampleMBean,
NotificationListener
{

  /**
   *
   * On service start, subscribes to notification sent by this
   * broadcaster or its remote peers.
   *
   */
  protected void startService() throws Exception
  {
    super.startService();
    addHANotificationListener(this);
  }

  /**
   *
   * On service stop, unsubscribes to notification sent by this
   * broadcaster or its remote peers.
   *
   */
  protected void stopService() throws Exception
  {
    removeHANotificationListener(this);
    super.stopService();
  }

  /**
   * Broadcasts a notification to the cluster partition.
   *
   * This example does not ensure that a notification sequence number
   * is unique throughout the partition.
   *
   */
  public void sendTextMessageViaHANBExample(String message)
    throws InstanceNotFoundException, MBeanException,
```

```
    ReflectionException
  {
    long now = System.currentTimeMillis();
    Notification notification =
      new Notification(
        "hanotification.example.counter",
        super.getServiceName(),
        now,
        now,
        message);
    server.invoke(
        broadcasterName_,
        "sendNotification",
        new Object[] { notification },
        new String[] { Notification.class.getName() }
        );
  }

  /**
   * Lists the notifications received on the cluster partition
   */
  public Collection getReceivedNotifications()
  {
    return messages_;
  }

  /**
   * @return the name of the broadcaster MBean
   */
  public String getHANotificationBroadcasterName()
  {
    return broadcasterName_ == null ? null :
          broadcasterName_.toString();
  }

  /**
   *
   * Sets the name of the broadcaster MBean.
   *
   * @param
   */
  public void setHANotificationBroadcasterName
```

```
       (String newBroadcasterName)
    throws InvalidParameterException
  {
    if (newBroadcasterName == null)
    {
      throw new InvalidParameterException
                  ("Broadcaster MBean must be specified");
    }
    try
    {
      broadcasterName_ = new ObjectName(newBroadcasterName);
    }
    catch (MalformedObjectNameException mone)
    {
      log.error("Broadcaster MBean Object Name is malformed", mone);
      throw new InvalidParameterException("Broadcaster MBean is not
                                          correctly formatted");
    }
  }

  protected void addHANotificationListener(NotificationListener
listener)
     throws InstanceNotFoundException
  {
    server.addNotificationListener(broadcasterName_, listener,
      /* no need for filter */ null,
      /* no handback object */ null);
  }

  protected void removeHANotificationListener
```

```
                (NotificationListener listener)
      throws InstanceNotFoundException, ListenerNotFoundException
  {
    server.removeNotificationListener(broadcasterName_, listener);
  }


  public void handleNotification(
    Notification notification,
    java.lang.Object handback)
  {
    messages_.add(notification.getMessage());
  }

  // Attributes ----------------------------------------------

  Collection messages_ = new LinkedList();

  /**
   * The broadcaster MBean that this class listens to and
   * delegates HA notifications to
   */
  ObjectName broadcasterName_ = null;

}
```

The key points of the code are highlighted. Notice how subscription to the cluster broadcaster is acomplished via the MBean server API. It is also important to note that the subscription is local, both the client and the MBean server reside in the same JVM. The invocation to sendNotification() is also local (in-JVM). The cluster broadcaster hides the implementation details of working together with its remote peers to deliver notifications throughout all nodes.

If you are not too much concerned with dependency on JBoss specific classes, then you can directly extend the cluster broadcaster class in which case all method calls will be direct instead of proxied through the MBean Server. Here is the code for an extended broadcaster:

```
public class HANotificationBroadcasterExample
  extends HAServiceMBeanSupport
  implements HANotificationBroadcasterExampleMBean
{

  /**
   *
   * On service start, subscribes to notification sent by this
broadcaster
   * or its remote peers.
   *
   */
  protected void startService() throws Exception
  {
    super.startService();
    addNotificationListener(listener_, /* no need for filter */ null,
                            /* no handback object */ null);
  }

  /**
   *
   * On service stop, unsubscribes to notification sent by this
   * broadcaster or its remote peers.
   *
   */
  protected void stopService() throws Exception
  {
    removeNotificationListener(listener_);
    super.stopService();
  }

  /**
   * Broadcasts a notification to the cluster partition.
   *
   * This example does not ensure that a notification sequence number
   * is unique throughout the partition.
   *
   */
  public void sendTextMessage(String message)
  {
    long now = System.currentTimeMillis();
    Notification notification =
      new Notification
```

```
        ("hanotification.example.counter", super.getServiceName(),
                            now, now, message);
      sendNotification(notification);
    }

    /**
     * Lists the notifications received on the cluster partition
     */
    public Collection getReceivedNotifications()
    {
      return messages_;
    }


    Collection messages_ = new LinkedList();

    NotificationListener listener_ = new NotificationListener()
    {
      public void handleNotification(Notification notification,
                                     java.lang.Object handback)
       {
         messages_.add( notification.getMessage() );
       }
    };

}
```

This class has very similar behavior as the previous one and if you look at its MBean view in the JMX console, you will notice that it has the same ReceivedNotifications attribute. The messaging operation is called sendTextMessage().

Chapter 12

12. Trouble Shooting and Limitations

Possible problems and limitations

First, are you a Windows user?

Windows (2000, XP, etc.) has an interesting feature called "Media Sense". This feature detects when a network cable is (un-)plugged from any NIC. When a cable is unplugged, Media Sense will automatically remove all IP routes associated with this NIC and restore them once the cable is re-plugged. This feature is enabled by default on Windows.

The problem with this, is that when the routes associated with the NIC are removed, traffic between local applications for example becomes impossible. For example, multicast packets send by a JBoss instance will no more be received by itself!

Microsoft officially provides a way to disable the Media Sense feature (**`http://support.microsoft.com/default.aspx?scid=KB;en-us;q239924`**, "How to Disable Media Sense for TCP/IP in Windows"). Nevertheless, while disabling the Media Sense feature will no more remove the IP routes

once a cable is unplugged, multicast communication is still impossible[10].

Consequently, to avoid any problem under Windows, you must modify the JGroups stack, so that the UDP protocol enables its "loopback" feature[11]:

```
…
  <mbean code="org.jboss.ha.framework.server.ClusterPartition"
         name="jboss:service=DefaultPartition">
    <attribute name="PartitionConfig">
      <Config>
        <!-- UDP: if you have a multihomed machine,
             set the bind_addr attribute to the appropriate NIC IP
address -->
        <!-- UDP: On Windows machines, because of the
             Media Sense feature being broken with multicast
             (even after disabling media sense)
             set the loopback attribute to true -->
        <UDP mcast_addr="228.1.2.3" mcast_port="45566"
             ip_ttl="64" ip_mcast="true"
             mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
             ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
             loopback="true" />
        <PING timeout="2000" num_initial_members="3"
              up_thread="false" down_thread="false" />
        <MERGE2 min_interval="5000" max_interval="10000" />
        <FD />
        <VERIFY_SUSPECT timeout="1500"
                        up_thread="false" down_thread="false" />
        <pbcast.STABLE desired_avg_gossip="20000"
                       up_thread="false" down_thread="false" />
        <pbcast.NAKACK gc_lag="50"
                       retransmit_timeout="300,600,1200,2400,4800"
                       up_thread="false" down_thread="false" />
```

[10] Adding a virtual loopback adaptor will **not** solve the problem as well as the multicast route will not be associated with the loopback adaptor but with the unplugged adaptor. And if you assign the multicast address to the loopback adaptor, packets will no more be sent out of the network.

[11] This feature is available as of JBoss 3.0.5.

```
        <UNICAST timeout="5000" window_size="100" min_threshold="10"
                 down_thread="false" />
        <FRAG frag_size="8192"
              down_thread="false" up_thread="false" />
        <pbcast.GMS join_timeout="5000" join_retry_timeout="2000"
                    shun="false" print_local_addr="true" />
        <pbcast.STATE_TRANSFER up_thread="false" down_thread="false"
/>
      </Config>
    </attribute>
  </mbean>
…
```

Trouble Shooting

- Make sure your network switch does not block the multicast IP ranges
- Make sure you have multicast enabled on your box. Here's some help for Linux: http://www.tldp.org/HOWTO/Multicast-HOWTO.html
- We have had problems running a clustered node with Win2K machines running VMWare 3.x. If you have VMWare installed on your machine, disable the VMWare Virtual Ethernet Adapters in the Device Manager
- RedHat Linux, by default, installs a firewall that prevents IP multicast packets from being distributed. Make sure you don't have this option installed or disable it.
- On Linux, you may have to add a route for multicast packets. The following command creates the route for multicast:

```
$ route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

If all else fails...

If all else fails then you must use a non-multicast communication stack for JGroups. Modify the "PartitionProperties" attribute to have the following JGroups communication stack. For TCPPING, put your own hosts and ports (host[port]) in for the *initial_hosts* parameter. DO NOT put a given node's own name in this list. So, each machine's config may have to be different. Check out the JGroups documentation for more configuration information: http://www.jgroups.org/.

Listing 10-12-1. Non-multicast JGroups config

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
         name="jboss:service=DefaultPartition">
  <mbean-ref-list name="SynchronizedMBeans">
    <mbean-ref-list-element>jboss:service=HASessionState</mbean-ref-
list-element>
    <mbean-ref-list-element>jboss:service=HAJNDI</mbean-ref-list-
element>
  </mbean-ref-list>
  <attribute name="PartitionProperties">
TCP(start_port=7800):TCPPING(initial_hosts=frodo[7800],gandalf[7800];
port_range=5;timeout=3000;num_initial_members=3;up_thread=true;down_t
hread=true):VERIFY_SUSPECT(timeout=1500;down_thread=false;up_thread=f
alse):pbcast.STABLE(desired_avg_gossip=20000;down_thread=false;up_thr
ead=false):pbcast.NAKACK(down_thread=true;up_thread=true;gc_lag=100;r
etransmit_timeout=3000):pbcast.GMS(join_timeout=5000;join_retry_timeo
ut=2000;shun=false;print_local_addr=false;down_thread=true;up_thread=
true)
  </attribute>
  </mbean>
```

Limitations

- State transfer that occurs when a new node joins a cluster could be improved. Mainly, if the partition stores a lot of state (for SFSB for example), the state transfer could take some time and block SFSB activity while state is exchanged. Some possibilities are already examined.

- In the current implementation, it is not possible to hot-deploy services based on HAPartition and that require to participate in the initial state transfer exchange that occurs when the partition starts (but it is possible to hot-deploy new HAPartitions and related services if it is performed in a single step).

13. Index

B

bean-load-balance-policy 54, 55, 57, 59

C

Cache Invalidation
- Architecture 85
- Bridges 95
- Commit Options 83
- Distributed cache 87
- Distributed locking 87
- Distributed transactions 87
- EJB Integration 88
- Invalidation Bridge 86
- InvalidationsTxGrouper 89
- JBossCluster-based Bridge 97
- JMS-based Bridge 95
- Overview 83
- synchronous invalidations 86
- Use Cases 98

cluster-service.xml 33, 42

D

discovery 19, 20, 30, 49, 50
Distributed Replicant Manager 109, 110
Distributed State Service 112

E

Entity Beans 20, 58, 59, 60

F

Fail-over 20
Farming 20, 81

H

HAPartition 34, 56, 104, 105, 106, 107, 109, 113, 114, 115, 119, 147
home-load-balance-policy 54, 55, 57, 59
homogeneous 49
Hot deploy 20
HTTP Session 65
- Apache 66, 67, 68, 69, 70, 71, 74
- distributable 78
- Do you really need replication? 67
- mod_jk 66, 67, 68, 69, 70, 71, 72, 73, 74, 77
- Tomcat Session Replication 74

J

jboss-services.xml 42
JGroups 31
JGroups 13, 23, 31
JGroups 33
JGroups 34
JGroups 34
JGroups 34
JGroups 103
JNDI 20, 21, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 56, 58, 114, 119

L

Limitations 143, 147
Linux 145
Load balancing 19
Load-Balance Policies 60
load-balancing 19, 20, 25, 27, 28, 41, 113

M

Message Driven Beans 60
multicast 31, 50, 145, 146, 147

N

NotificationService 132

O

Other Clustering Services 121

P

partition 21, 22, 23, 24, 25, 28, 30, 33, 34, 43, 45, 54, 56, 57, 59, 81, 82, 104, 106, 147
proxies 25, 29
proxy 28, 55

R

RMI 20, 27, 28, 29, 44, 110, 113, 114
RPC 31, 105, 106, 107, 108

S

scalability 19, 23
Scheduler Service 129
Singleton Service 121
State transfer 108, 119, 147
Stateful Session Bean 55
Stateful Session Beans 20, 24, 55
Stateless Session Beans 20, 54
Synchronization 59

T

TCPPING 146, 147
Trouble Shooting 143, 145

V

VMWare 145

W

Windows
Media-Sense 143