HornetQ 2.0 User Manual

Putting the buzz in messaging

Table of Contents

1

Legal Notice

2

Preface

What is HornetQ?

- HornetQ is an open source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.
- HornetQ is an example of Message Oriented Middleware (MoM) For a description of MoMs and other messaging concepts please see the Chapter 4.
- For answers to more questions about what HornetQ is and what it isn't please visit the FAQs wiki page [http://www.jboss.org/community/wiki/HornetQGeneralFAQs].

Why use HornetQ? Here are just a few of the reasons:

- 100% open source software. HornetQ is licenced using the Apache Software License v 2.0 to minimise barriers to adoption.
- HornetQ is designed with usability in mind.
- Written in Java. Runs on any platform with a Java 5+ runtime, that's everything from Windows desktops to IBM mainframes.
- Amazing performance. Our ground-breaking high performance journal provides persistent messaging performance at rates normally seen for non-persistent messaging, our non-persistent messaging performance rocks the boat too.
- Full feature set. All the features you'd expect in any serious messaging system, and others you won't find anywhere else.
- Elegant, clean-cut design with minimal third party dependencies. Run HornetQ stand-alone, run it in integrated in your favourite JEE application server, or run it embedded inside your own product. It's up to you.
- Seamless high availability. We provide a HA solution with automatic client failover so you can guarantee zero message loss or duplication in event of server failure.
- Hugely flexible clustering. Create clusters of servers that know how to load balance messages. Link geographically distributed clusters over unreliable connections to form a global network. Configure routing of messages in a highly flexible way.
- For a full list of features, please see the features wiki page [http://www.jboss.org/community/wiki/HornetQFeatures] .

3

Project Information

The official HornetQ project page is http://hornetq.org/.

3.1. Software Download

The software can be download from the Download page:http://hornetq.org/downloads.html

3.2. Project Information

- Please take a look at our project wiki [http://www.jboss.org/community/wiki/HornetQ]
- If you have any user questions please use our user forum [http://www.jboss.org/index.html?module=bb&op=viewforum&f=312]
- If you have development related questions, please use our developer forum [http://www.jboss.org/index.html?module=bb&op=viewforum&f=313]
- Pop in and chat to us in our IRC channel [irc://irc.freenode.net:6667/hornetq]
- Our project blog [http://hornetq.blogspot.com/]
- Follow us on twitter [http://twitter.com/hornetq]
- HornetQ Subversion trunk is http://anonsvn.jboss.org/repos/hornetq/trunk
- All release tags are available from http://anonsvn.jboss.org/repos/hornetq/tags

Red Hat kindly employs developers to work full time on HornetQ, they are:

- Tim Fox [http://jbossfox.blogspot.com] (project lead)
- Howard Gao
- Jeff Mesnil [http://jmesnil.net/weblog/]
- Clebert Suconic
- Andy Taylor

And many thanks to all our contributors, both old and new who helped create HornetQ, for a full list of the people who made it happen, take a look at our team page [http://jboss.org/hornetq/community/team.html].

4

Messaging Concepts

HornetQ is an asynchronous messaging system, an example of Message Oriented Middleware [http://en.wikipedia.org/wiki/Message_oriented_middleware] , we'll just call them messaging systems in the remainder of this book.

We'll first present a brief overview of what kind of things messaging systems do, where they're useful and the kind of concepts you'll hear about in the messaging world.

If you're already familiar with what a messaging system is and what it's capable of, then you can skip this chapter.

4.1. Messaging Concepts

Messaging systems allow you to loosely couple heteregenous systems together, whilst typically providing reliability, transactions and many other features.

Unlike systems based on a Remote Procedure Call [http://en.wikipedia.org/wiki/Remote_procedure_call] (RPC) pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode but this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end-to-end allows you to really take advantage of your hardware resources, minimizing the amount of threads blocking on IO operations, and to use your network bandwidth to its full capacity. With an RPC approach you have to wait for a response for each request you make so are limited by the network round trip time, or *latency* of your network. With an asynchronous system you can pipeline flows of messages in different directions, so are limited by the network *bandwidth* not the latency. This typically allows you to create much higher performance applications.

Messaging systems decouple the senders of messages from the consumers of messages. The senders and consumers of messages are completely independent and know nothing of each other. This allows you to create flexible, loosely coupled systems.

Often, large enterprises use a messaging system to implement a message bus which loosely couples heterogeneous systems together. Message buses often form the core of an Enterprise Service Bus [http://en.wikipedia.org/wiki/Enterprise_service_bus]. (ESB). Using a message bus to de-couple disparate systems can allow the system to grow and adapt more easily. It also allows more flexibility to add new systems or retire old ones since they don't have brittle dependencies on each other.

4.2. Messaging styles

Messaging systems normally support two main styles of asynchronous messaging: message queue

[http://en.wikipedia.org/wiki/Message_queue] messaging (also known as *point-to-point messaging*) and publish subscribe [http://en.wikipedia.org/wiki/Publish_subscribe] messaging. We'll summarise them briefly here:

4.2.1. The Message Queue Pattern

With this type of messaging you send a message to a queue. The message is then typically persisted to provide a guarantee of delivery, then some time later the messaging system delivers the message to a consumer. The consumer then processes the message and when it is done, it acknowledges the message. Once the message is acknowledged it disappears from the queue and is not available to be delivered again. If the system crashes before the messaging server receives an acknowledgement from the consumer, then on recovery, the message will be available to be delivered to a consumer again.

With point-to-point messaging, there can be many consumers on the queue but a particular message will only ever be consumed by a maximum of one of them. Senders (also known as *producers*) to the queue are completely decoupled from receivers (also known as *consumers*) of the queue - they do not know of each others existence.

A classic example of point to point messaging would be an order queue in a company's book ordering system. Each order is represented as a message which is sent to the order queue. Let's imagine there are many front end ordering systems which send orders to the order queue. When a message arrives on the queue it is persisted - this ensures that if the server crashes the order is not lost. Let's also imagine there are many consumers on the order queue - each representing an instance of an order processing component - these can be on different physical machines but consuming from the same queue. The messaging system delivers each message to one and only one of the ordering processing components. Different messages can be processed by different order processors, but a single order is only processed by one order processor - this ensures orders aren't processed twice.

As an order processor receives a message, it fulfills the order, sends order information to the warehouse system and then updates the order database with the order details. Once it's done that it acknowledges the message to tell the server that the order has been processed and can be forgotten about. Often the send to the warehouse system, update in database and acknowledgement will be completed in a single transaction to ensure ACID [http://en.wikipedia.org/wiki/ACID] properties.

4.2.2. The Publish-Subscribe Pattern

With publish-subscribe messaging many senders can send messages to an entity on the server, often called a *topic* (e.g. in the JMS world).

There can be many *subscriptions* on a topic, a subscription is just another word for a consumer of a topic. Each subscription receives a *copy* of *each* message sent to the topic. This differs from the message queue pattern where each message is only consumed by a single consumer.

Subscriptions can optionally be *durable* which means they retain a copy of each message sent to the topic until the subscriber consumes them - even if the server crashes or is restarted in between. Non-durable subscriptions only last a maximum of the lifetime of the connection that created them.

An example of publish-subscribe messaging would be a news feed. As news articles are created by different editors around the world they are sent to a news feed topic. There are many subscribers around the world who are interested in receiving news items - each one creates a subscription and the messaging system ensures that a copy of each news message is delivered to each subscription.

4.3. Delivery guarantees

A key feature of most messaging systems is *reliable messaging*. With reliable messaging the server gives a guarantee that the message will be delivered once and only once to each consumer of a queue or each durable subscription of a topic, even in the event of system failure. This is crucial for many businesses; e.g. you don't want your orders fulfilled more than once or any of your orders to be lost.

In other cases you may not care about a once and only once delivery guarantee and are happy to cope with duplicate deliveries or lost messages - an example of this might be transient stock price updates - which are quickly superseded by the next update on the same stock. The messaging system allows you to configure which delivery guarantees you require.

4.4. Transactions

Messaging systems typically support the sending and acknowledgement of multiple messages in a single local transaction. HornetQ also supports the sending and acknowledgement of message as part of a large global transaction - using the Java mapping of XA; JTA.

4.5. Durability

Messages are either durable or non durable. Durable messages will be persisted in permanent storage and will survive server failure or restart. Non durable messages will not survive server failure or restart. Examples of durable messages might be orders or trades, where they cannot be lost. An example of a non durable message might be a stock price update which is transitory and doesn't need to survive a restart.

4.6. Messaging APIs and protocols

How do client applications interact with messaging systems in order to send and consume messages?

Several messaging systems provide their own proprietary APIs with which the client communicates with the messaging system.

There are also some standard ways of operating with messaging systems and some emerging standards in this space.

Let's take a brief look at these:

4.6.1. Java Message Service (JMS)

JMS [http://en.wikipedia.org/wiki/Java_Message_Service] is part of Sun's JEE specification. It's a Java API that encapsulates both message queue and publish-subscribe messaging patterns. JMS is a lowest common denominator specification - i.e. it was created to encapsulate common functionality of the already existing messaging systems that were available at the time of its creation.

JMS is a very popular API and is implemented by most, messaging systems. JMS is only available to clients run-

ning Java.

JMS does not define a standard wire format - it only defines a programmatic API so JMS clients and servers from different vendors cannot directly interoperate since each will use the vendor's own internal wire protocol.

HornetQ provides a fully compliant JMS 1.1 API.

4.6.2. System specific APIs

Many systems provide their own programmatic API for which to interact with the messaging system. The advantage of this it allows the full set of system functionality to be exposed to the client application. API's like JMS are not normally rich enough to expose all the extra features that most messaging systems provide.

HornetQ provides its own core client API for clients to use if they wish to have access to functionality over and above that accessible via the JMS API.

4.6.3. RESTful API

REST [http://en.wikipedia.org/wiki/Representational_State_Transfer] approaches to messaging are showing a lot interest recently.

It seems plausible that API standards for cloud computing may converge on a REST style set of interfaces and consequently a REST messaging approach is a very strong contender for becoming the defacto method for messaging interoperability.

With a REST approach messaging resources are manipulated as resources defined by a URI and typically using a simple set of operations on those resources, e.g. PUT, POST, GET etc. REST approaches to messaging often use HTTP as their underlying protocol.

The advantage of a REST approach with HTTP is in its simplicity and the fact the internet is already tuned to deal with HTTP optimally.

HornetQ will shortly be implementing RESTful approach to messaging interoperability.

4.6.4. STOMP

STOMP [http://en.wikipedia.org/wiki/Streaming_Text_Orientated_Messaging_Protocol] is a very simple protocol for interoperating with messaging systems. It defines a wire format, so theoretically any STOMP client can work with any messaging system that supports STOMP. STOMP clients are available in many different programming languages.

HornetQ can be used by any STOMP client when using the StompConnect [http://stomp.codehaus.org/StompConnect] broker which translates the STOMP protocol to the JMS API.

HornetQ will be shortly implementing the STOMP protocol on the broker, thus avoiding having to use StompConnect.

4.6.5. AMQP

AMQP [http://en.wikipedia.org/wiki/AMQP] is a specification for interoperable messaging. It also defines a wire format, so any AMQP client can work with any messaging system that supports AMQP. AMQP clients are available in many different programming languages.

HornetQ will shortly be implementing AMQP.

4.7. High Availability

High Availability (HA) means that the system should remain operational after failure of one or more of the servers. The degree of support for HA varies between various messaging systems.

HornetQ provides automatic failover where your sessions are automatically reconnected to the backup server on event of live serve failure.

For more information on HA, please see Chapter 39.

4.8. Clusters

Many messaging systems allow you to create groups of messaging servers called *clusters*. Clusters allow the load of sending and consuming messages to be spread over many servers. This allows your system to scale horizontally by adding new servers to the cluster.

Degrees of support for clusters varies between messaging systems, with some systems having fairly basic clusters with the cluster members being hardly aware of each other.

HornetQ provides very configurable state-of-the-art clustering model where messages can be intelligently load balanced between the servers in the cluster, according to the number of consumers on each node, and whether they are ready for messages.

HornetQ also has the ability to automatically redistribute messages between nodes of a cluster to prevent starvation on any particular node.

For full details on clustering, please see Chapter 38.

4.9. Bridges and routing

Some messaging systems allow isolated clusters or single nodes to be bridged together, typically over unreliable connections like a wide area network (WAN), or the internet.

A bridge normally consumes from a queue on one server and forwards messages to another queue on a different server. Bridges cope with unreliable connections, automatically reconnecting when the connections becomes available again.

HornetQ bridges can be configured with filter expressions to only forward certain messages, and transformation can also be hooked in.

HornetQ also allows routing between queues to be configured in server side configuration. This allows complex

routing networks to be set up forwarding or copying messages from one destination to another, forming a global network of interconnected brokers.

For more information please see Chapter 36 and Chapter 35.

5

Architecture

In this section we will give an overview of the HornetQ high level architecture.

5.1. Core Architecture

HornetQ core is designed simply as set of Plain Old Java Objects (POJOs) - we hope you like it's clean-cut design.

We've also designed it to have as few dependencies on external jars as possible. In fact, HornetQ core has only one jar dependency, netty.jar, other than the standard JDK classes! This is because we use some of the netty buffer classes internally.

This allows HornetQ to be easily embedded in your own project, or instantiated in any dependency injection framework such as JBoss Microcontainer, Spring or Google Guice.

Each HornetQ server has its own ultra high performance persistent journal, which it uses for message and other persistence.

Using a high performance journal allows outrageous persistence message performance, something not achievable when using a relational database for persistence.

HornetQ clients, potentially on different physical machines interact with the HornetQ server. HornetQ currently provides two APIs for messaging at the client side:

1. Core client API. This is a simple intuitive Java API that allows the full set of messaging functionality without some of the complexities of JMS.
2. JMS client API. The standard JMS API is available at the client side.

JMS semantics are implemented by a thin JMS facade layer on the client side.

The HornetQ server does not speak JMS and in fact does not know anything about JMS, it's a protocol agnostic messaging server designed to be used with multiple different protocols.

When a user uses the JMS API on the client side, all JMS interactions are translated into operations on the HornetQ core client API before being transferred over the wire using the HornetQ wire format.

The server always just deals with core API interactions.

A schematic illustrating this relationship is shown in figure 3.1 below:

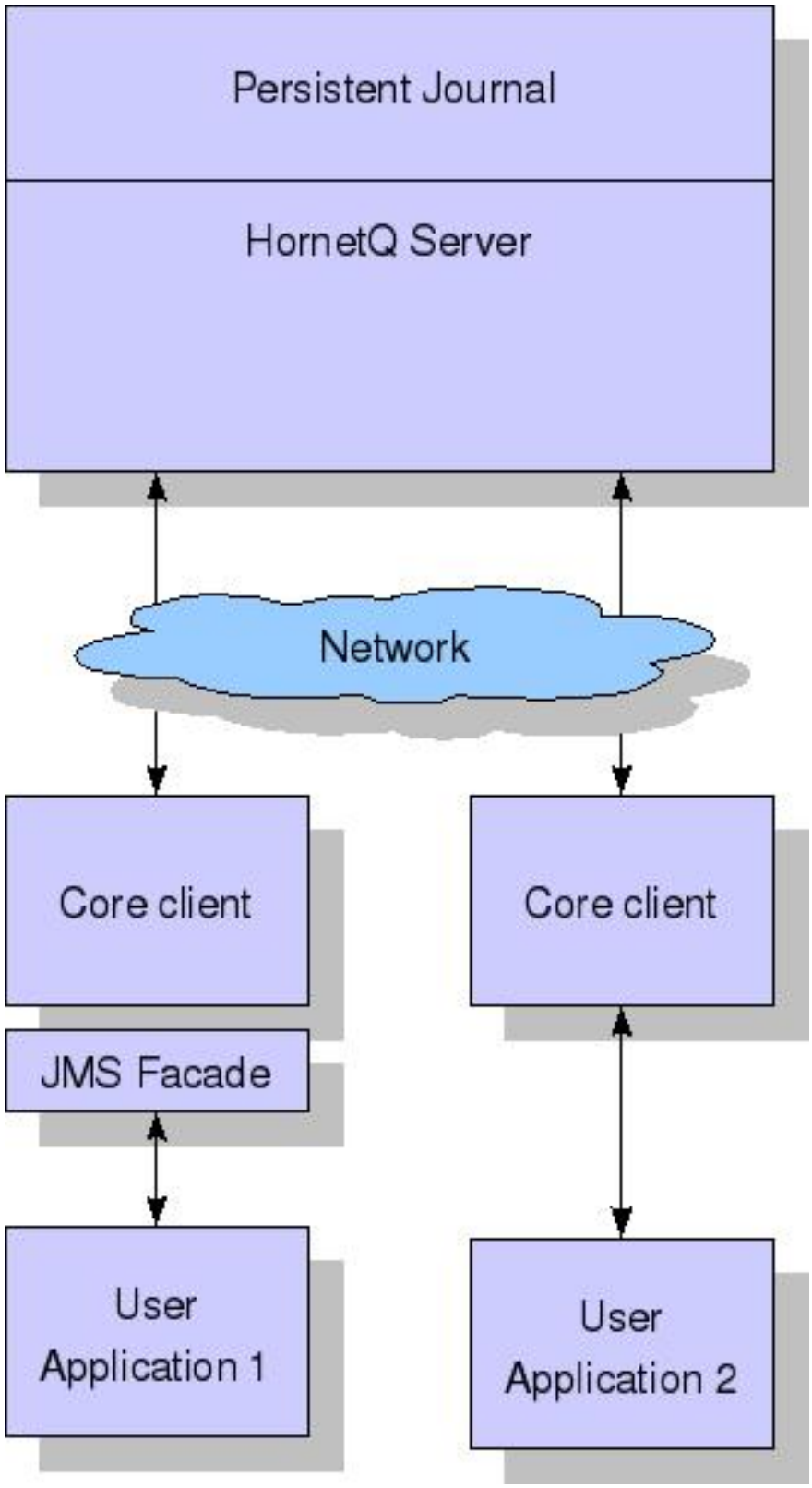


Figure 3.1 shows two user applications interacting with a HornetQ server. User Application 1 is using the JMS API, while User Application 2 is using the core client API directly.

You can see from the diagram that the JMS API is implemented by a thin facade layer on the client side.

5.2. HornetQ embedded in your own application

HornetQ core is designed as a set of simple POJOs so if you have an application that requires messaging functionality internally but you don't want to expose that as a HornetQ server you can directly instantiate and embed HornetQ servers in your own application.

For more information on embedding HornetQ, see Chapter 43.

5.3. HornetQ integrated with a JEE application server

HornetQ provides its own fully functional Java Connector Architecture (JCA) adaptor which enables it to be integrated easily into any JEE compliant application server or servlet engine.

JEE application servers provide Message Driven Beans (MDBs), which are a special type of Enterprise Java Beans (EJBs) that can process messages from sources such as JMS systems or mail systems.

Probably the most common use of an MDB is to consume messages from a JMS messaging system.

According to the JEE specification, a JEE application server uses a JCA adapter to integrate with a JMS messaging system so it can consume messages for MDBs.

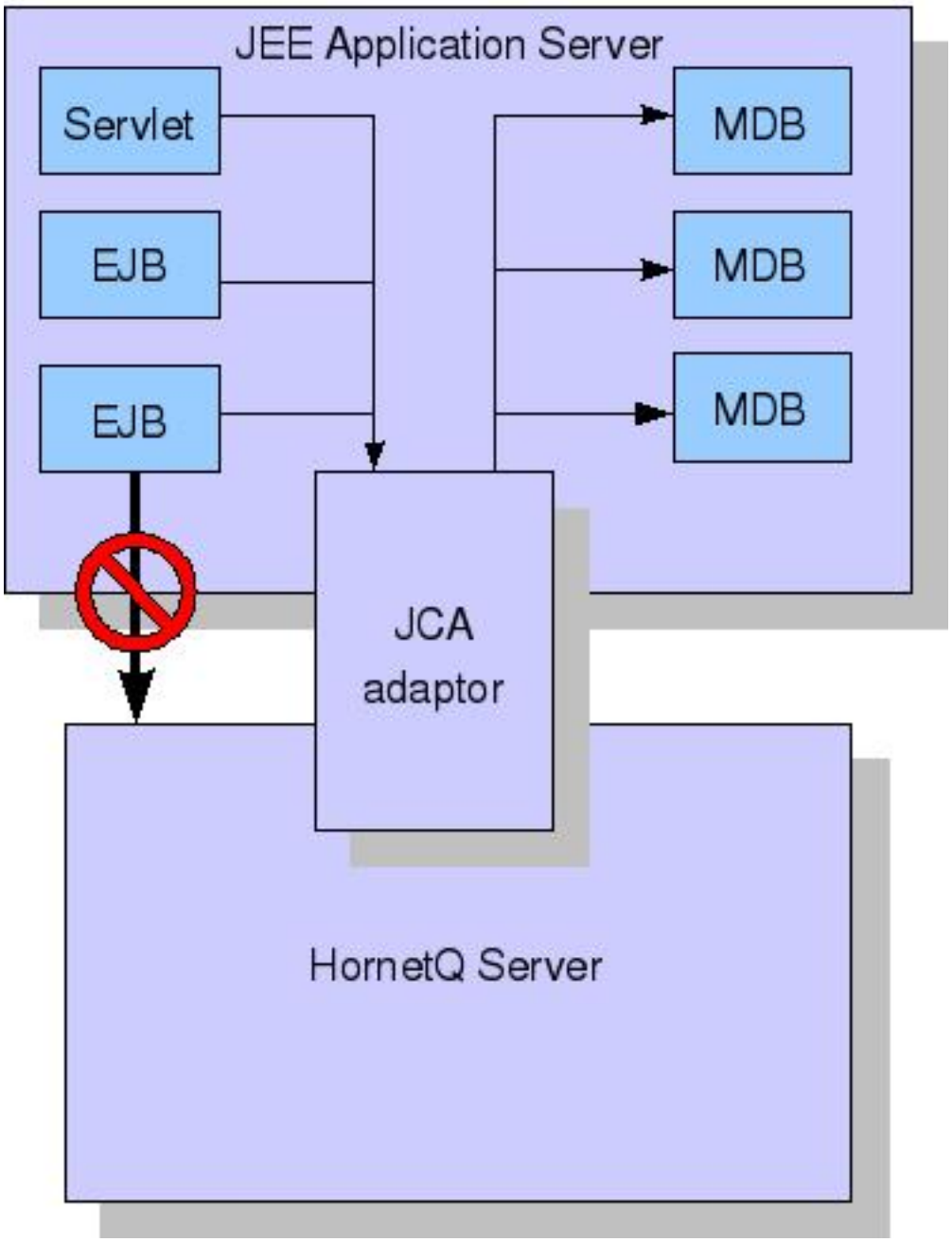
However, the JCA adapter is not only used by the JEE application server for *consuming* messages via MDBs, it is also used when sending message to the JMS messaging system e.g. from inside an EJB or servlet.

When integrating with a JMS messaging system from inside a JEE application server it is always recommended that this is done via a JCA adaptor. In fact, communicating with a JMS messaging system directly, without using JCA would be illegal according to the JEE specification.

The application server's JCA service provides extra functionality such as connection pooling and automatic transaction enlistment, which are desirable when using messaging, say, from inside an EJB. It is possible to talk to a JMS messaging system directly from an EJB, MDB or servlet without going through a JCA adapter, but this is not recommended since you will not be able to take advantage of the JCA features, such as caching of JMS sessions, which can result in poor performance.

Figure 3.2 below shows a JEE application server integrating with a HornetQ server via the HornetQ JCA adaptor. Note that all communication between EJB sessions or entity beans and Message Driven beans go through the adaptor and not directly to HornetQ.

The large arrow with the prohibited sign shows an EJB session bean talking directly to the HornetQ server. This is not recommended as you'll most likely end up creating a new connection and session every time you want to interact from the EJB, which is an anti-pattern.



For more information on using the JCA adaptor, please see Chapter 32.

5.4. HornetQ stand-alone server

HornetQ can also be deployed as a stand-alone server. This means a fully independent messaging server not dependent on a JEE application server.

The standard stand-alone messaging server configuration comprises a core messaging server, a JMS service and a JNDI service.

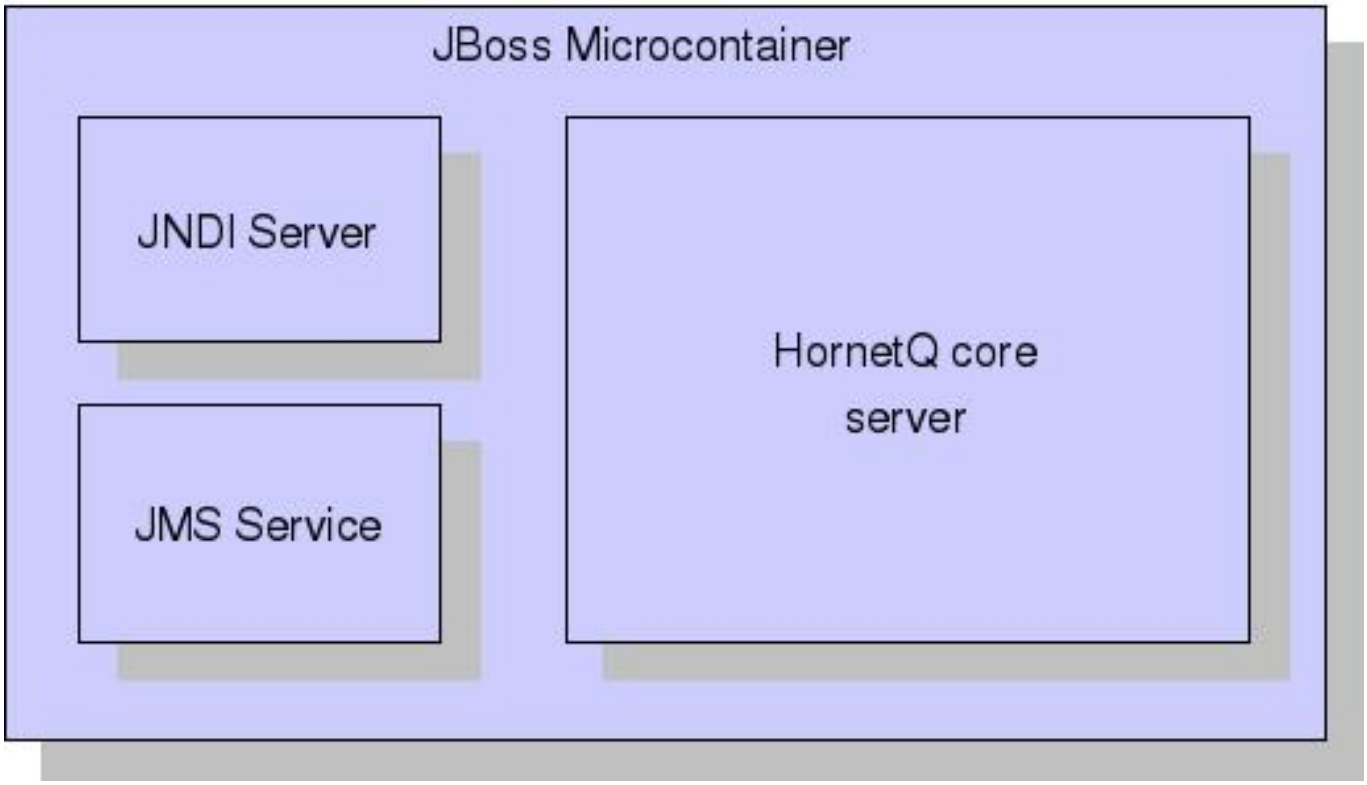
The role of the JMS Service is to deploy any JMS Queue, Topic and ConnectionFactory instances from any server side `hornetq-jms.xml` configuration files. It also provides a simple management API for creating and destroying Queues, Topics and ConnectionFactory instances which can be accessed via JMX or the connection. It is a separate

service to the HornetQ core server, since the core server is JMS agnostic. If you don't want to deploy any JMS Queue, Topic or ConnectionFactory instances via server side XML configuration and don't require a JMS management API on the server side then you can disable this service.

We also include a JNDI server since JNDI is a common requirement when using JMS to lookup Queues, Topics and ConnectionFactory instances. If you do not require JNDI then this service can also be disabled. HornetQ allows you to programmatically create JMS and core objects directly on the client side as opposed to looking them up from JNDI, so a JNDI server is not always a requirement.

The stand-alone server configuration uses JBoss Microcontainer to instantiate and enforce dependencies between the components. JBoss Microcontainer is a very lightweight POJO bootstrapper.

The stand-alone server architecture is shown in figure 3.3 below:



For more information on server configuration files see Section 47.1. $

6

Using the Server

This chapter will familiarise you with how to use the HornetQ server.

We'll show where it is, how to start and stop it, and we'll describe the directory layout and what all the files are and what they do.

For the remainder of this chapter when we talk about the HornetQ server we mean the HornetQ standalone server, in its default configuration with a JMS Service and JNDI service enabled.

When running embedded in JBoss Application Server the layout may be slightly different but by-and-large will be the same.

6.1. Starting and Stopping the standalone server

In the distribution you will find a directory called `bin`.

`cd` into that directory and you'll find a unix/linux script called `run.sh` and a windows batch file called `run.bat`

To run on Unix/Linux type `./run.sh`

To run on Windows type `run.bat`

These scripts are very simple and basically just set-up the classpath and some JVM parameters and start the JBoss Microcontainer. The Microcontainer is a light weight container used to deploy the HornetQ POJO's

To stop the server you'll also find a unix/linux script `stop.sh` and a windows batch file `run.bat`

To run on Unix/Linux type `./stop.sh`

To run on Windows type `stop.bat`

Please note that HornetQ requires a Java 5 or later runtime to run. We recommend running on Java 6.

Both the run and the stop scripts use the config under `config/stand-alone/non-clustered` by default. The configuration can be changed by running `./run.sh ../config/stand-alone/clustered` or another config of your choosing. This is the same for the stop script and the windows bat files.

6.2. Server JVM settings

The run scripts `run.sh` and `run.bat` set some JVM settings for tuning running on Java 6 and choosing the garbage collection policy. We recommend using a parallel garbage collection algorithm to smooth out latency and minimise

large GC pauses.

By default HornetQ runs in a maximum of 1GiB of RAM. To increase the memory settings change the `-Xms` and `-Xmx` memory settings as you would for any Java program.

If you wish to add any more JVM arguments or tune the existing ones, the run scripts are the place to do it.

6.3. Server classpath

HornetQ looks for its configuration files on the Java classpath.

The scripts `run.sh` and `run.bat` specify the classpath when calling Java to run the server.

In the distribution, the run scripts will add the non clustered configuration directory to the classpath. This is a directory which contains a set of configuration files for running the HornetQ server in a basic non-clustered configuration. In the distribution this directory is `config/stand-alone/non-clustered/` from the root of the distribution.

The distribution contains several standard configuration sets for running:

- Non clustered stand-alone.
- Clustered stand-alone
- Non clustered in JBoss Application Server
- Clustered in JBoss Application Server

You can of course create your own configuration and specify any configuration directory when running the run script.

Just make sure the directory is on the classpath and HornetQ will search there when starting up.

6.4. Library Path

If you're using the Asynchronous IO Journal on Linux, you need to specify `java.library.path` as a property on your Java options. This is done automatically in the `run.sh` script.

If you don't specify `java.library.path` at your Java options then the JVM will use the environment variable `LD_LIBRARY_PATH`.

6.5. System properties

HornetQ can take a system property on the command line for configuring logging.

For more information on configuring logging, please see Chapter 42.

6.6. Configuration files

The configuration directory is specified on the classpath in the run scripts `run.sh` and `run.bat` This directory can contain the following files.

- `hornetq-beans.xml` (or `hornetq-jboss-beans.xml` if you're running inside JBoss Application Server). This is the JBoss Microcontainer beans file which defines what beans the Microcontainer should create and what dependencies to enforce between them. Remember that HornetQ is just a set of POJOs. In the stand-alone server, it's the JBoss Microcontainer which instantiates these POJOs and enforces dependencies between them and other beans.

- `hornetq-configuration.xml`. This is the main HornetQ configuration file. All the parameters in this file are described in Chapter 47. Please see Section 6.9 for more information on this file.

- `hornetq-queues.xml`. This file contains predefined queues, queue settings and security settings. The file is optional - all this configuration can also live in `hornetq-configuration.xml`. In fact, the default configuration sets do not have a `hornetq-queues.xml` file. The purpose of allowing queues to be configured in these files is to allow you to manage your queue configuration over many files instead of being forced to maintain it in a single file. There can be many `hornetq-queues.xml` files on the classpath. All will be loaded if found.

- `hornetq-users.xml` HornetQ ships with a basic security manager implementation which obtains user credentials from the `hornetq-users.xml` file. This file contains user, password and role information. For more information on security ,please see Chapter 31.

- `hornetq-jms.xml` The distro configuration by default includes a server side JMS service which mainly deploys JMS Queues, Topics and ConnectionFactorys from this file into JNDI. If you're not using JMS, or you don't need to deploy JMS objects on the server side, then you don't need this file. For more information on using JMS, please see Chapter 7.

- `logging.properties` This is used to configure the logging handlers used by the Java logger. For more information on configuring logging, please see Chapter 42.

- `log4j.xml` This is the Log4j configuration if the Log4j handler is configured.

Note

The property `file-deployment-enabled` in the `hornetq-configuration.xml` configuration when set to false means that the other configuration files are not loaded. This is true by default.

It is also possible to use system property substitution in all the configuration files. by replacing a value with the name of a system property. Here is an example of this with a connector configuration:

```
<connector name="netty">
         <factory-class>org.hornetq.integration.transports.netty.NettyConnectorFactory
           </factory-class>
         <param key="host"  value="${hornetq.remoting.netty.host:localhost}" type="String"/>
         <param key="port"  value="${hornetq.remoting.netty.port:5445}" type="Integer"/>
</connector>
```

Here you can see we have replaced 2 values with system properties `hornetq.remoting.netty.host` and `hornetq.remoting.netty.port`. These values will be replaced by the value found in the system property if there is one, if not they default back to localhost or 5445 respectively. It is also possible to not supply a default. i.e. `${hornetq.remoting.netty.host}`, however the system property *must* be supplied in that case.

6.7. JBoss Microcontainer Beans File

The stand-alone server is basically a set of POJOs which are instantiated by the light weight JBoss Microcontainer [http://www.jboss.org/jbossmc/]engine.

Note

A beans file is also needed when the server is deployed in the JBoss Application Server but this will deploy a slightly different set of objects since the Application Server will already have things like security etc deployed.

Let's take a look at an example beans file from the stand-alone server:

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

<bean name="Naming" class="org.jnp.server.NamingBeanImpl"/>

<!-- JNDI server. Disable this if you don't want JNDI -->
<bean name="JNDIServer" class="org.jnp.server.Main">
   <property name="namingInfo">
      <inject bean="Naming"/>
   </property>
   <property name="port">1099</property>
   <property name="bindAddress">localhost</property>
   <property name="rmiPort">1098</property>
   <property name="rmiBindAddress">localhost</property>
</bean>

<!-- MBean server -->
<bean name="MBeanServer" class="javax.management.MBeanServer">
   <constructor factoryClass="java.lang.management.ManagementFactory"
      factoryMethod="getPlatformMBeanServer"/>
</bean>

<!-- The core configuration -->
<bean name="Configuration" class="org.hornetq.core.config.impl.FileConfiguration">
</bean>

<!-- The security manager -->
<bean name="HornetQSecurityManager"
      class="org.hornetq.spi.core.security.HornetQSecurityManagerImpl">
   <start ignored="true"/>
   <stop ignored="true"/>
</bean>

<!-- The core server -->
<bean name="HornetQServer" class="org.hornetq.core.server.impl.HornetQServerImpl">
   <start ignored="true"/>
   <stop ignored="true"/>
   <constructor>
      <parameter>
         <inject bean="Configuration"/>
      </parameter>
      <parameter>
         <inject bean="MBeanServer"/>
      </parameter>
      <parameter>
         <inject bean="HornetQSecurityManager"/>
```

```
      </parameter>
   </constructor>
</bean>

<!-- The JMS server -->
<bean name="JMSServerManager"
      class="org.hornetq.jms.server.impl.JMSServerManagerImpl">
   <constructor>
      <parameter>
         <inject bean="HornetQServer"/>
      </parameter>
   </constructor>
</bean>

</deployment>
```

We can see that, as well as the core HornetQ server, the stand-alone server instantiates various different POJOs, lets look at them in turn:

- JNDIServer

  Many clients like to look up JMS Objects from JNDI so we provide a JNDI server for them to do that. If you don't need JNDI this can be commented out or removed.

- MBeanServer

  In order to provide a JMX management interface a JMS MBean server is necessary in which to register the management objects. Normally this is just the default platform MBean server available in the JVM instance. If you don't want to provide a JMX management interface this can be commented out or removed.

- Configuration

  The HornetQ server is configured with a Configuration object. In the default stand-alone set-up it uses a FileConfiguration object which knows to read configuration information from the file system. In different configurations such as embedded you might want to provide configuration information from somewhere else.

- Security Manager. The security manager used by the messaging server is pluggable. The default one used just reads user-role information from the `hornetq-users.xml` file on disk. However it can be replaced by a JAAS security manager, or when running inside JBoss Application Server it can be configured to use the JBoss AS security manager for tight integration with JBoss AS security. If you've disabled security altogether you can remove this too.

- HornetQServer

  This is the core server. It's where 99% of the magic happens

- JMSServerManager

  This deploys any JMS Objects such as JMS Queues, Topics and ConnectionFactory instances from `hornetq-jms.xml` files on the disk. It also provides a simple management API for manipulating JMS Objects. On the whole it just translates and delegates its work to the core server. If you don't need to deploy JMS Queues, Topics and ConnectionFactorys from server side configuration and don't require the JMS management interface this can be disabled.

6.8. JBoss AS4 MBean Service.

Note

The section is only to configure HornetQ on JBoss AS4. The service funtionality is similar to Microcontainer Beans

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

   <mbean code="org.hornetq.service.HornetQFileConfigurationService"
      name="org.hornetq:service=HornetQFileConfigurationService">
   </mbean>

   <mbean code="org.hornetq.service.JBossASSecurityManagerService"
      name="org.hornetq:service=JBossASSecurityManagerService">
   </mbean>

   <mbean code="org.hornetq.service.HornetQStarterService"
      name="org.hornetq:service=HornetQStarterService">
      <!--lets let the JMS Server start us-->
         <attribute name="Start">false</attribute>

      <depends optional-attribute-name="SecurityManagerService"
         proxy-type="attribute">org.hornetq:service=JBossASSecurityManagerService</depends>
      <depends optional-attribute-name="ConfigurationService"
         proxy-type="attribute">org.hornetq:service=HornetQFileConfigurationService</depends>
   </mbean>

   <mbean code="org.hornetq.service.HornetQJMSStarterService"
      name="org.hornetq:service=HornetQJMSStarterService">
      <depends optional-attribute-name="HornetQServer"
         proxy-type="attribute">org.hornetq:service=HornetQStarterService</depends>
   </mbean>

</server>
```

This jboss-service.xml configuration file is included inside the hornetq-service.sar on AS4 with embebbed HornetQ. As you can see, on this configuration file we are starting various services:

- HornetQFileConfigurationService

  This is an MBean Service that takes care of the life cycle of the `FileConfiguration POJO`

- JBossASSecurityManagerService

  This is an MBean Service that takes care of the lifecycle of the `JBossASSecurityManager` POJO

- HornetQStarterService

  This is an MBean Service that controls the main `HornetQServer` POJO. this has a dependency on JBossASSecurityManagerService and HornetQFileConfigurationService MBeans

- HornetQJMSStarterService

This is an MBean Service that controls the `JMSServerManagerImpl` POJO. If you aren't using jms this can be removed.

- JMSServerManager

  Has the responsibility to start the JMSServerManager and the same behaviour that JMSServerManager Bean

6.9. The main configuration file.

The configuration for the HornetQ core server is contained in `hornetq-configuration.xml`. This is what the FileConfiguration bean uses to configure the messaging server.

There are many attributes which you can configure HornetQ. In most cases the defaults will do fine, in fact every attribute can be defaulted which means a file with a single empty `configuration` element is a valid configuration file. The different configuration will be explained throughout the manual or you can refer to the configuration reference here.

7

Using JMS

Although HornetQ provides a JMS agnostic messaging API, many users will be more comfortable using JMS.

JMS is a very popular API standard for messaging, and most messaging systems provide a JMS API. If you are completely new to JMS we suggest you follow the Sun JMS tutorial [http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html] - a full JMS tutorial is out of scope for this guide.

HornetQ also ships with a wide range of examples, many of which demonstrate JMS API usage. A good place to start would be to play around with the simple JMS Queue and Topic example, but we also provide examples for many other parts of the JMS API. A full description of the examples is available in Chapter 11.

In this section we'll go through the main steps in configuring the server for JMS and creating a simple JMS program. We'll also show how to configure and use JNDI, and also how to use JMS with HornetQ without using any JNDI.

7.1. A simple ordering system

For this chapter we're going to use a very simple ordering system as our example. It's a somewhat contrived example because of its extreme simplicity, but it serves to demonstrate the very basics of setting up and using JMS.

We will have a single JMS Queue called `OrderQueue`, and we will have a single `MessageProducer` sending an order message to the queue and a single `MessageConsumer` consuming the order message from the queue.

The queue will be a `durable` queue, i.e. it will survive a server restart or crash. We also want to predeploy the queue, i.e. specify the queue in the server JMS configuration so it's created automatically without us having to explicitly create it from the client.

7.2. JMS Server Configuration

The file `hornetq-jms.xml` on the server classpath contains any JMS Queue, Topic and ConnectionFactory instances that we wish to create and make available to lookup via the JNDI.

A JMS ConnectionFactory object is used by the client to make connections to the server. It knows the location of the server it is connecting to, as well as many other configuration parameters. In most cases the defaults will be acceptable.

We'll deploy a single JMS Queue and a single JMS Connection Factory instance on the server for this example but there are no limits to the number of Queues, Topics and Connection Factory instances you can deploy from the file. Here's our configuration:

```
<configuration xmlns="urn:hornetq"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:hornetq ../schemas/hornetq-jms.xsd ">

    <connection-factory name="ConnectionFactory">
        <connectors>
           <connector-ref connector-name="netty"/>
        </connectors>
        <entries>
            <entry name="ConnectionFactory"/>
        </entries>
    </connection-factory>

    <queue name="OrderQueue">
        <entry name="queues/OrderQueue"/>
    </queue>

</configuration>
```

We deploy one ConnectionFactory called `ConnectionFactory` and bind it in just one place in JNDI as given by the `entry` element. ConnectionFactory instances can be bound in many places in JNDI if you require.

Note

The JMS connection factory references a `connector` called `netty`. This is a reference to a connector object deployed in the main core configuration file `hornetq-configuration.xml` which defines the transport and parameters used to actually connect to the server.

7.3. JNDI configuration

When using JNDI from the client side you need to specify a set of JNDI properties which tell the JNDI client where to locate the JNDI server, amongst other things. These are often specified in a file called `jndi.properties` on the client classpath, or you can specify them directly when creating the JNDI initial context. A full JNDI tutorial is outside the scope of this document, please see the Sun JNDI tutorial [http://java.sun.com/products/jndi/tutorial/TOC.html] for more information on how to use JNDI.

For talking to the JBoss JNDI Server, the jndi properties will look something like this:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://myhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Where `myhost` is the hostname or IP address of the JNDI server. 1099 is the port used by the JNDI server and may vary depending on how you have configured your JNDI server.

In the default standalone configuration, JNDI server ports are configured in the file `hornetq-beans.xml` by setting properties on the `JNDIServer` bean:

```
<bean name="JNDIServer" class="org.jnp.server.Main">
    <property name="namingInfo">
```

```
        <inject bean="Naming"/>
    </property>
    <property name="port">1099</property>
    <property name="bindAddress">localhost</property>
    <property name="rmiPort">1098</property>
    <property name="rmiBindAddress">localhost</property>
</bean>
```

Note

If you want your JNDI server to be available to non local clients make sure you change it's bind address to something other than `localhost`!

Note

The JNDIServer bean must be defined *only when HornetQ is running in stand-alone mode*. When HornetQ is integrated to JBoss Application Server, JBoss AS will provide a ready-to-use JNDI server without any additional configuration.

7.4. The code

Here's the code for the example:

First we'll create a JNDI initial context from which to lookup our JMS objects:

```
InitialContect ic = new InitialContext();
```

Now we'll look up the connection factory:

```
ConnectionFactory cf = (ConnectionFactory)ic.lookup("/ConnectionFactory");
```

And look up the Queue:

```
Queue orderQueue = (Queue)ic.lookup("/queues/OrderQueue");
```

Next we create a JMS connection using the connection factory:

```
Connection connection = cf.createConnection();
```

And we create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

We create a MessageProducer that will send orders to the queue:

```
MessageProducer producer = session.createProducer(orderQueue);
```

And we create a MessageConsumer which will consume orders from the queue:

```
MessageConsumer consumer = session.createConsumer(orderQueue);
```

We make sure we start the connection, or delivery won't occur on it:

```
connection.start();
```

We create a simple TextMessage and send it:

```
TextMessage message = session.createTextMessage("This is an order");
producer.send(message);
```

And we consume the message:

```
TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

It's as simple as that. For a wide range of working JMS examples please see the examples directory in the distribution.

Warning

Please note that JMS connections, sessions, producers and consumers are *designed to be re-used*.

It's an anti-pattern to create new connections, sessions, producers and consumers for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning Chapter 46.

7.5. Directly instantiating JMS Resources without using JNDI

Although it's a very common JMS usage pattern to lookup JMS *Administered Objects* (that's JMS Queue, Topic and ConnectionFactory instances) from JNDI, in some cases a JNDI server is not available and you still want to use JMS, or you just think "Why do I need JNDI? Why can't I just instantiate these objects directly?"

With HornetQ you can do exactly that. HornetQ supports the direct instantiation of JMS Queue, Topic and ConnectionFactory instances, so you don't have to use JNDI at all.

For a full working example of direct instantiation please see the JMS examples in Chapter 11.

Here's our simple example, rewritten to not use JNDI at all:

We create the JMS ConnectionFactory object via the HornetQJMSClient Utility class, note we need to provide connection parameters and specify which transport we are using, for more information on connectors please see Chapter 16.

```
TransportConfiguration transportConfiguration =
                      new TransportConfiguration(NettyConnectorFactory.class.getName());
ConnectionFactory cf = HornetQJMSClient.createConnectionFactory(transportConfiguration);
```

We also create the JMS Queue object via the HornetQJMSClient Utility class:

```
Queue orderQueue = HornetQJMSClient.createQueue("OrderQueue");
```

Next we create a JMS connection using the connection factory:

```
Connection connection = cf.createConnection();
```

And we create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

We create a MessageProducer that will send orders to the queue:

```
MessageProducer producer = session.createProducer(orderQueue);
```

And we create a MessageConsumer which will consume orders from the queue:

```
MessageConsumer consumer = session.createConsumer(orderQueue);
```

We make sure we start the connection, or delivery won't occur on it:

```
connection.start();
```

We create a simple TextMessage and send it:

```
TextMessage message = session.createTextMessage("This is an order");
producer.send(message);
```

And we consume the message:

```
TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

7.6. Setting The Client ID

This represents the client id for a JMS client and is needed for creating durable subscriptions. It is possible to configure this on the connection factory and can be set via the `client-id` element. Any connection created by this connection factory will have this set as its client id.

7.7. Setting The Batch Size for DUPS_OK

When the JMS acknowledge mode is set to `DUPS_OK` it is possible to configure the consumer so that it sends acknowledgements in batches rather that one at a time, saving valuable bandwidth. This can be configured via the connection factory via the `dups-ok-batch-size` element and is set in bytes. The default is 1024 * 1024 bytes = 1 MiB.

7.8. Setting The Transaction Batch Size

When receiving messages in a transaction it is possible to configure the consumer to send acknowledgements in

batches rather than individually saving valuable bandwidth. This can be configured on the connection factory via the `transaction-batch-size` element and is set in bytes. The default is 1024 * 1024.

8

Using Core

HornetQ core is a completely JMS-agnostic messaging system with its own non-JMS API. We call this the *core API*.

If you don't want to use JMS you can use the core API directly. The core API provides all the functionality of JMS but without much of the complexity. It also provides features that are not available using JMS.

8.1. Core Messaging Concepts

Some of the core messaging concepts are similar to JMS concepts, but core messaging concepts differ in some ways. In general the core messaging API is simpler than the JMS API, since we remove distinctions between queues, topics and subscriptions. We'll discuss each of the major core messaging concepts in turn, but to see the API in detail, please consult the Javadoc.

8.1.1. Message

- A message is the unit of data which is sent between clients and servers.
- A message has a body which is a buffer containing convenient methods for reading and writing data into it.
- A message has a set of properties which are key-value pairs. Each property key is a string and property values can be of type integer, long, short, byte, byte[], String, double, float or boolean.
- A message has an *address* it is being sent to. When the message arrives on the server it is routed to any queues that are bound to the address - if the queues are bound with any filter, the message will only be routed to that queue if the filter matches. An address may have many queues bound to it or even none. There may also be entities other than queues, like *diverts* bound to addresses.
- Messages can be either durable or non durable. Durable messages in a durable queue will survive a server crash or restart. Non durable messages will never survive a server crash or restart.
- Messages can be specified with a priority value between 0 and 9. 0 represents the lowest priority and 9 represents the highest. HornetQ will attempt to deliver higher priority messages before lower priority ones.
- Messages can be specified with an optional expiry time. HornetQ will not deliver messages after its expiry time has been exceeded.
- Messages also have an optional timestamp which represents the time the message was sent.
- HornetQ also supports the sending/consuming of very large messages - much larger than can fit in available

RAM at any one time.

8.1.2. Address

A server maintains a mapping between an address and a set of queues. Zero or more queues can be bound to a single address. Each queue can be bound with an optional message filter. When a message is routed, it is routed to the set of queues bound to the message's address. If any of the queues are bound with a filter expression, then the message will only be routed to the subset of bound queues which match that filter expression.

Other entities, such as *diverts* can also be bound to an address and messages will also be routed there.

Note

In core, there is no concept of a Topic, Topic is a JMS only term. Instead, in core, we just deal with *addresses* and *queues*.

For example, a JMS topic would implemented by a single address to which many queues are bound. Each queue represents a subscription of the topic. A JMS Queue would be implemented as a single address to which one queue is bound - that queue represents the JMS queue.

8.1.3. Queue

Queues can be durable, meaning the messages they contain survive a server crash or restart, as long as the messages in them are durable. Non durable queues do not survive a server restart or crash even if the messages they contain are durable.

Queues can also be temporary, meaning they are automatically deleted when the client connection is closed, if they are not explicitly deleted before that.

Queues can be bound with an optional filter expression. If a filter expression is supplied then the server will only route messages that match that filter expression to any queues bound to the address.

Many queues can be bound to a single address. A particular queue is only bound to a maximum of one address.

8.1.4. ClientSessionFactory

Clients use `ClientSessionFactory` instances to create `ClientSession` instances. `ClientSessionFactory` instances know how to connect to the server to create sessions, and are configurable with many settings.

`ClientSessionFactory` instances are created using the `HornetQClient` factory class.

8.1.5. ClientSession

A client uses a ClientSession for consuming and producing messages and for grouping them in transactions. ClientSession instances can support both transactional and non transactional semantics and also provide an `XAResource` interface so messaging operations can be performed as part of a JTA [http://java.sun.com/javaee/technologies/jta/index.jsp] transaction.

ClientSession instances group ClientConsumers and ClientProducers.

ClientSession instances can be registered with an optional `SendAcknowledgementHandler`. This allows your client code to be notified asynchronously when sent messages have successfully reached the server. This unique HornetQ feature, allows you to have full guarantees that sent messages have reached the server without having to block on each message sent until a response is received. Blocking on each messages sent is costly since it requires a network round trip for each message sent. By not blocking and receiving send acknowledgements asynchronously you can create true end to end asynchronous systems which is not possible using the standard JMS API. For more information on this advanced feature please see the section Chapter 20.

8.1.6. ClientConsumer

Clients use `ClientConsumer` instances to consume messages from a queue. Core Messaging supports both synchronous and asynchronous message consumption semantics. `ClientConsumer` instances can be configured with an optional filter expression and will only consume messages which match that expression.

8.1.7. ClientProducer

Clients create `ClientProducer` instances on `ClientSession` instances so they can send messages. ClientProducer instances can specify an address to which all sent messages are routed, or they can have no specified address, and the address is specified at send time for the message.

Warning

Please note that ClientSession, ClientProducer and ClientConsumer instances are *designed to be re-used*.

It's an anti-pattern to create new ClientSession, ClientProducer and ClientConsumer instances for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning Chapter 46.

8.2. A simple example of using Core

Here's a very simple program using the core messaging API to send and receive a message:

```
ClientSessionFactory factory =  HornetQClient.createClientSessionFactory(
                                       new TransportConfiguration(
                                          InVMConnectorFactory.class.getName()));

ClientSession session = factory.createSession();

session.createQueue("example", "example", true);

ClientProducer producer = session.createProducer("example");

ClientMessage message = session.createMessage(true);

message.getBodyBuffer().writeString("Hello");

producer.send(message);

session.start();
```

```
ClientConsumer consumer = session.createConsumer("example");

ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBodyBuffer().readString());

session.close();
```

9

Mapping JMS Concepts to the Core API

This chapter describes how JMS destinations are mapped to HornetQ addresses.

HornetQ core is JMS-agnostic. It does not have any concept of a JMS topic. A JMS topic is implemented in core as an address (the topic name) with zero or more queues bound to it. Each queue bound to that address represents a topic subscription. Likewise, a JMS queue is implemented as an address (the JMS queue name) with one single queue bound to it which represents the JMS queue.

By convention, all JMS queues map to core queues where the core queue name has the string `jms.queue.` prepended to it. E.g. the JMS queue with the name "orders.europe" would map to the core queue with the name "jms.queue.orders.europe". The address at which the core queue is bound is also given by the core queue name.

For JMS topics the address at which the queues that represent the subscriptions are bound is given by prepending the string "jms.topic." to the name of the JMS topic. E.g. the JMS topic with name "news.europe" would map to the core address "jms.topic.news.europe"

In other words if you send a JMS message to a JMS queue with name "orders.europe" it will get routed on the server to any core queues bound to the address "jms.queue.orders.europe". If you send a JMS message to a JMS topic with name "news.europe" it will get routed on the server to any core queues bound to the address "jms.topic.news.europe".

If you want to configure settings for a JMS Queue with the name "orders.europe", you need to configure the corresponding core queue "jms.queue.orders.europe":

```
<!-- expired messages in JMS Queue "orders.europe"
    will be sent to the JMS Queue "expiry.europe" -->
<address-setting match="jms.queue.orders.europe">
   <expiry-address>jms.queue.expiry.europe</expiry-address>
   ...
</address-setting>
```

10

The Client Classpath

HornetQ requires several jars on the *Client Classpath* depending on whether the client uses HornetQ Core API, JMS, and JNDI.

Warning

All the jars mentioned here can be found in the `lib` directory of the HornetQ distribution. Be sure you only use the jars from the correct version of the release, you *must not* mix and match versions of jars from different HornetQ versions. Mixing and matching different jar versions may cause subtle errors and failures to occur.

10.1. HornetQ Core Client

If you are using just a pure HornetQ Core client (i.e. no JMS) then you need `hornetq-core-client.jar`, `hornetq-transports.jar` and `netty.jar` on your client classpath.

10.2. JMS Client

If you are using JMS on the client side, then you will also need to include `hornetq-jms-client.jar` and `jboss-jms-api.jar`.

Note

`jboss-jms-api.jar` just contains Java EE API interface classes needed for the `javax.jms.*` classes. If you already have a jar with these interface classes on your classpath, you will not need it.

10.3. JMS Client with JNDI

If you are looking up JMS resources from the JNDI server co-located with the HornetQ standalone server, you wil also need the jar `jnp-client.jar` jar on your client classpath as well as any other jars mentioned previously.

11

Examples

The HornetQ distribution comes with over 70 run out-of-the-box examples demonstrating many of the features.

The examples are available in the distribution, in the `examples` directory. Examples are split into JMS and core examples. JMS examples show how a particular feature can be used by a normal JMS client. Core examples show how the equivalent feature can be used by a core messaging client.

A set of Java EE examples are also provided which need the JBoss Application Server installed to be able to run.

11.1. JMS Examples

To run a JMS example, simply `cd` into the appropriate example directory and type `./build.sh` (or `build.bat` if you are on Windows).

Here's a listing of the examples with a brief description.

11.1.1. Application-Layer Failover

HornetQ also supports Application-Layer failover, useful in the case that replication is not enabled on the server side.

With Application-Layer failover, it's up to the application to register a JMS `ExceptionListener` with HornetQ which will be called by HornetQ in the event that connection failure is detected.

The code in the `ExceptionListener` then recreates the JMS connection, session, etc on another node and the application can continue.

Application-layer failover is an alternative approach to High Availability (HA). Application-layer failover differs from automatic failover in that some client side coding is required in order to implement this. Also, with Application-layer failover, since the old session object dies and a new one is created, any uncommitted work in the old session will be lost, and any unacknowledged messages might be redelivered.

11.1.2. Core Bridge Example

The `bridge` example demonstrates a core bridge deployed on one server, which consumes messages from a local queue and forwards them to an address on a second server.

Core bridges are used to create message flows between any two HornetQ servers which are remotely separated. Core bridges are resilient and will cope with temporary connection failure allowing them to be an ideal choice for forwarding over unreliable connections, e.g. a WAN.

11.1.3. Browser

The `browser` example shows you how to use a JMS `QueueBrowser` with HornetQ.

Queues are a standard part of JMS, please consult the JMS 1.1 specification for full details.

A `QueueBrowser` is used to look at messages on the queue without removing them. It can scan the entire content of a queue or only messages matching a message selector.

11.1.4. Client Kickoff

The `client-kickoff` example shows how to terminate client connections given an IP address using the JMX management API.

11.1.5. Client-Side Load-Balancing

The `client-side-load-balancing` example demonstrates how sessions created from a single JMS `Connection` can be created to different nodes of the cluster. In other words it demonstrates how HornetQ does client-side load-balancing of sessions across the cluster.

11.1.6. Clustered Grouping

This is similar to the message grouping example except that it demonstrates it working over a cluster. Messages sent to different nodes with the same group id will be sent to the same node and the same consumer.

11.1.7. Clustered Queue

The `clustered-queue` example demonstrates a JMS queue deployed on two different nodes. The two nodes are configured to form a cluster. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

11.1.8. Clustered Standalone

The `clustered-standalone` example demonstrates how to configure and starts 3 cluster nodes on the same machine to form a cluster. A subscriber for a JMS topic is created on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that the 3 subscribers receive all the sent messages.

11.1.9. Clustered Topic

The `clustered-topic` example demonstrates a JMS topic deployed on two different nodes. The two nodes are configured to form a cluster. We then create a subscriber on the topic on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both subscribers receive all the sent messages.

11.1.10. Message Consumer Rate Limiting

With HornetQ you can specify a maximum consume rate at which a JMS MessageConsumer will consume messages. This can be specified when creating or deploying the connection factory.

If this value is specified then HornetQ will ensure that messages are never consumed at a rate higher than the specified rate. This is a form of consumer throttling.

11.1.11. Dead Letter

The `dead-letter` example shows you how to define and deal with dead letter messages. Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back).

Such a message goes back to the JMS destination ready to be redelivered. However, this means it is possible for a message to be delivered again and again without any success and remain in the destination, clogging the system.

To prevent this, messaging systems define dead letter messages: after a specified unsuccessful delivery attempts, the message is removed from the destination and put instead in a dead letter destination where they can be consumed for further investigation.

11.1.12. Delayed Redelivery

The `delayed-redelivery` example demonstrates how HornetQ can be configured to provide a delayed redelivery in the case a message needs to be redelivered.

Delaying redelivery can often be useful in the case that clients regularly fail or roll-back. Without a delayed redelivery, the system can get into a "thrashing" state, with delivery being attempted, the client rolling back, and delivery being re-attempted in quick succession, using up valuable CPU and network resources.

11.1.13. Divert

HornetQ diverts allow messages to be transparently "diverted" or copied from one address to another with just some simple configuration defined on the server side.

11.1.14. Durable Subscription

The `durable-subscription` example shows you how to use a durable subscription with HornetQ. Durable subscriptions are a standard part of JMS, please consult the JMS 1.1 specification for full details.

Unlike non-durable subscriptions, the key function of durable subscriptions is that the messages contained in them persist longer than the lifetime of the subscriber - i.e. they will accumulate messages sent to the topic even if there is no active subscriber on them. They will also survive server restarts or crashes. Note that for the messages to be persisted, the messages sent to them must be marked as durable messages.

11.1.15. Embedded

The `embedded` example shows how to embed the HornetQ server within your own code.

11.1.16. HTTP Transport

The `http-transport` example shows you how to configure HornetQ to use the HTTP protocol as its transport layer.

11.1.17. Instantiate JMS Objects Directly

Usually, JMS Objects such as `ConnectionFactory`, `Queue` and `Topic` instances are looked up from JNDI before being used by the client code. This objects are called "administered objects" in JMS terminology.

However, in some cases a JNDI server may not be available or desired. To come to the rescue HornetQ also supports the direct instantiation of these administered objects on the client side so you don't have to use JNDI for JMS.

11.1.18. Interceptor

HornetQ allows an application to use an interceptor to hook into the messaging system. Interceptors allow you to handle various message events in HornetQ.

11.1.19. JAAS

The `jaas` example shows you how to configure HornetQ to use JAAS for security. HornetQ can leverage JAAS to delegate user authentication and authorization to existing security infrastructure.

11.1.20. JMS Bridge

The `jms-brige` example shows how to setup a bridge between two standalone HornetQ servers.

11.1.21. JMX Management

The `jmx` example shows how to manage HornetQ using JMX.

11.1.22. Large Message

The `large-message` example shows you how to send and receive very large messages with HornetQ. HornetQ supports the sending and receiving of huge messages, much larger than can fit in available RAM on the client or server. Effectively the only limit to message size is the amount of disk space you have on the server.

Large messages are persisted on the server so they can survive a server restart. In other words HornetQ doesn't just do a simple socket stream from the sender to the consumer.

11.1.23. Last-Value Queue

The `last-value-queue` example shows you how to define and deal with last-value queues. Last-value queues are special queues which discard any messages when a newer message with the same value for a well-defined last-value property is put in the queue. In other words, a last-value queue only retains the last value.

A typical example for last-value queue is for stock prices, where you are only interested by the latest price for a particular stock.

11.1.24. Load Balanced Clustered Queue

The `clustered-queue` example demonstrates a JMS queue deployed on two different nodes. The two nodes are configured to form a cluster.

We then create a consumer on the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

In other words, HornetQ load balances the sent messages across all consumers on the cluster

11.1.25. Management

The `management` example shows how to manage HornetQ using JMS Messages to invoke management operations on the server.

11.1.26. Management Notification

The `management-notification` example shows how to receive management notifications from HornetQ using JMS messages. HornetQ servers emit management notifications when events of interest occur (consumers are created or closed, addresses are created or deleted, security authentication fails, etc.).

11.1.27. Message Counter

The `message-counters` example shows you how to use message counters to obtain message information for a JMS queue.

11.1.28. Message Expiration

The `expiry` example shows you how to define and deal with message expiration. Messages can be retained in the messaging system for a limited period of time before being removed. JMS specification states that clients should not receive messages that have been expired (but it does not guarantee this will not happen).

HornetQ can assign an expiry address to a given queue so that when messages are expired, they are removed from the queue and sent to the expiry address. These "expired" messages can later be consumed from the expiry address for further inspection.

11.1.29. Message Group

The `message-group` example shows you how to configure and use message groups with HornetQ. Message groups allow you to pin messages so they are only consumed by a single consumer. Message groups are sets of messages that has the following characteristics:

- Messages in a message group share the same group id, i.e. they have same JMSXGroupID string property values
- The consumer that receives the first message of a group will receive all the messages that belongs to the group

11.1.30. Message Group

The `message-group2` example shows you how to configure and use message groups with HornetQ via a connection factory.

11.1.31. Message Priority

Message Priority can be used to influence the delivery order for messages.

It can be retrieved by the message's standard header field 'JMSPriority' as defined in JMS specification version 1.1.

The value is of type integer, ranging from 0 (the lowest) to 9 (the highest). When messages are being delivered, their priorities will effect their order of delivery. Messages of higher priorities will likely be delivered before those of lower priorities.

Messages of equal priorities are delivered in the natural order of their arrival at their destinations. Please consult the JMS 1.1 specification for full details.

11.1.32. No Consumer Buffering

By default, HornetQ consumers buffer messages from the server in a client side buffer before you actually receive them on the client side. This improves performance since otherwise every time you called receive() or had processed the last message in a `MessageListener onMessage()` method, the HornetQ client would have to go the server to request the next message, which would then get sent to the client side, if one was available.

This would involve a network round trip for every message and reduce performance. Therefore, by default, HornetQ pre-fetches messages into a buffer on each consumer.

In some case buffering is not desirable, and HornetQ allows it to be switched off. This example demonstrates that.

11.1.33. Non-Transaction Failover With Server Data Replication

The `non-transaction-failover` example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a *non-transacted* JMS session failing over from live to backup when the live server is crashed.

HornetQ implements failover of client connections between live and backup servers. This is implemented by the replication of state between live and backup nodes. When replication is configured and a live node crashes, the client connections can carry and continue to send and consume messages. When non-transacted sessions are used, once and only once message delivery is not guaranteed and it is possible that some messages will be lost or delivered twice.

11.1.34. Paging

The `paging` example shows how HornetQ can support huge queues even when the server is running in limited RAM. It does this by transparently *paging* messages to disk, and *depaging* them when they are required.

11.1.35. Pre-Acknowledge

Standard JMS supports three acknowledgement modes: `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`. For a full description on these modes please consult the JMS specification, or any JMS tutorial.

All of these standard modes involve sending acknowledgements from the client to the server. However in some cases, you really don't mind losing messages in event of failure, so it would make sense to acknowledge the message on the server before delivering it to the client. This example demonstrates how HornetQ allows this with an extra acknowledgement mode.

11.1.36. Message Producer Rate Limiting

The `producer-rte-limit` example demonstrates how, with HornetQ, you can specify a maximum send rate at which a JMS message producer will send messages.

11.1.37. Queue

A simple example demonstrating a JMS queue.

11.1.38. Message Redistribution

The `queue-message-redistribution` example demonstrates message redistribution between queues with the same name deployed in different nodes of a cluster.

11.1.39. Queue Requestor

A simple example demonstrating a JMS queue requestor.

11.1.40. Queue with Message Selector

The `queue-selector` example shows you how to selectively consume messages using message selectors with queue consumers.

11.1.41. Reattach Node example

The `Reattach Node` example shows how a client can try to reconnect to the same server instead of failing the connection immediately and notifying any user ExceptionListener objects. HornetQ can be configured to automatically retry the connection, and reattach to the server when it becomes available again across the network.

11.1.42. Request-Reply example

A simple example showing the JMS request-response pattern.

11.1.43. Scheduled Message

The `scheduled-message` example shows you how to send a scheduled message to a JMS Queue with HornetQ. Scheduled messages won't get delivered until a specified time in the future.

11.1.44. Security

The `security` example shows you how configure and use role based queue security with HornetQ.

11.1.45. Send Acknowledgements

The `send-acknowledgements` example shows you how to use HornetQ's advanced *asynchronous send acknowledgements* feature to obtain acknowledgement from the server that sends have been received and processed in a separate stream to the sent messages.

11.1.46. SSL Transport

The `ssl-enabled` shows you how to configure SSL with HornetQ to send and receive message.

11.1.47. Static Message Selector

The `static-selector` example shows you how to configure a HornetQ core queue with static message selectors (filters).

11.1.48. Static Message Selector Using JMS

The `static-selector-jms` example shows you how to configure a HornetQ queue with static message selectors (filters) using JMS.

11.1.49. Symmetric Cluster

The `symmetric-cluster` example demonstrates a symmetric cluster set-up with HornetQ.

HornetQ has extremely flexible clustering which allows you to set-up servers in many different topologies. The most common topology that you'll perhaps be familiar with if you are used to application server clustering is a symmetric cluster.

With a symmetric cluster, the cluster is homogeneous, i.e. each node is configured the same as every other node, and every node is connected to every other node in the cluster.

11.1.50. Temporary Queue

A simple example demonstrating how to use a JMS temporary queue.

11.1.51. Topic

A simple example demonstrating a JMS topic.

11.1.52. Topic Hierarchy

HornetQ supports topic hierarchies. With a topic hierarchy you can register a subscriber with a wild-card and that subscriber will receive any messages sent to an address that matches the wild card.

11.1.53. Topic Selector 1

The `topic-selector-example1` example shows you how to send message to a JMS Topic, and subscribe them using selectors with HornetQ.

11.1.54. Topic Selector 2

The `topic-selector-example1` example shows you how to selectively consume messages using message selectors with topic consumers.

11.1.55. Transaction Failover With Data Replication

The `transaction-failover` example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a transacted JMS session failing over from live to backup when the live server is crashed.

HornetQ implements failover of client connections between live and backup servers. This is implemented by the replication of data between live and backup nodes. When replication is configured and a live node crashes, the client connections can carry and continue to send and consume messages. When transacted sessions are used, once and only once message delivery is guaranteed.

11.1.56. Transactional Session

The `transactional` example shows you how to use a transactional Session with HornetQ.

11.1.57. XA Heuristic

The `xa-heuristic` example shows you how to make an XA heuristic decision through HornetQ Management Interface. A heuristic decision is a unilateral decision to commit or rollback an XA transaction branch after it has been prepared.

11.1.58. XA Receive

The `xa-receive` example shows you how message receiving behaves in an XA transaction in HornetQ.

11.1.59. XA Send

The `xa-send` example shows you how message sending behaves in an XA transaction in HornetQ.

11.1.60. XA with Transaction Manager

The `xa-with-jta` example shows you how to use JTA interfaces to control transactions with HornetQ.

11.2. Core API Examples

To run a core example, simply `cd` into the appropriate example directory and type `ant`

11.2.1. Embedded

This example shows how to embed the HornetQ server within your own code.

11.3. Java EE Examples

Most of the Java EE examples can be run the following way. simply cd into the appropriate example directory an type `ant deploy`. This will create a new JBoss AS profile and start the server. When the server is started from a different window type `ant run` to run the example. Some examples require further steps, please refer to the examples documentation for further instructions.

11.3.1. EJB/JMS Transaction

An example that shows using an EJB and JMS together within a transaction.

11.3.2. HAJNDI (High Availability)

A simple example demonstrating using JNDI within a cluster.

11.3.3. Resource Adapter Configuration

This example demonstrates how to configure several properties on the HornetQ JCA resource adaptor.

11.3.4. JMS Bridge

An example demonstrating the use of the HornetQ JMS bridge.

11.3.5. MDB (Message Driven Bean)

A simple example of a message driven bean.

11.3.6. Servlet Transport

An example of how to use the HornetQ servlet transport.

11.3.7. Servlet SSL Transport

An example of how to use the HornetQ servlet transport over SSL.

11.3.8. XA Recovery

An example of how XA recovery works within the JBoss Application server using HornetQ.

12

Routing Messages With Wild Cards

HornetQ allows the routing of messages via wildcard addresses.

If a consumer is created with an address of say `queue.news.#` then it will receive any messages sent to addresses that match this, for instance `queue.news.europe` or `queue.news.usa` or `queue.news.usa.sport`. This allows a consumer to consume messages which are sent to a *hierarchy* of addresses, rather than the consumer having to specify a specific address.

Note

In JMS terminology this allows "topic hierarchies" to be created.

To enable this functionality set the property `wild-card-routing-enabled` in the `hornetq-configuration.xml` file to `true`. This is `true` by default.

For more information on the wild card syntax take a look at Chapter 13 chapter, also see Section 11.1.52.

13

Understanding the HornetQ Wildcard Syntax

HornetQ uses a specific syntax for representing wildcards in security settings, address settings and when creating consumers.

The syntax is similar to that used by AMQP [www.amqp.org].

A HornetQ wildcard expression contains words delimited by the character '.' (full stop).

The special characters '#' and '*' also have special meaning and can take the place of a word.

The character '#' means 'match any sequence of zero or more words'.

The character '*' means 'match a single word'.

So the wildcard 'news.europe.#' would match 'news.europe', 'news.europe.sport', 'news.europe.politics', and 'news.europe.politics.regional' but would not match 'news.usa', 'news.usa.sport' nor 'entertainment'.

The wildcard 'news.*' would match 'news.europe', but not 'news.europe.sport'.

The wildcard 'news.*.sport' would match 'news.europe.sport' and also 'news.usa.sport', but not 'news.europe.politics'.

14

Filter Expressions

HornetQ provides a powerful filter language based on a subset of the SQL 92 expression syntax.

It is the same as the syntax used for JMS selectors, but the predefined identifiers are different. For documentation on JMS selector syntax please the JMS javadoc for javax.jms.Message [http://java.sun.com/javaee/5/docs/api/javax/jms/Message.html].

Filter expressions are used in several places in HornetQ

- Predefined Queues. When pre-defining a queue, either in `hornetq-configuration.xml` or `hornetq-jms.xml` a filter expression can be defined for a queue. Only messages that match the filter expression will enter the queue.
- Core bridges can be defined with an optional filter expression, only matching messages will be bridged (see Chapter 36).
- Diverts can be defined with an optional filter expression, only matching messages will be diverted (see Chapter 35).
- Filter are also used programmatically when creating consumers, queues and in several places as described in Chapter 30.

There are some differences between JMS selector expressions and HornetQ core filter expressions. Whereas JMS selector expressions operate on a JMS message, HornetQ core filter expressions operate on a core message.

The following identifiers can be used in a core filter expressions to refer to attributes of the core message in an expression:

- `HQPriority`. To refer to the priority of a message. Message priorities are integers with valid values from `0 - 9`. `0` is the lowest priority and `9` is the highest. E.g. `HQPriority = 3 AND animal = 'aardvark'`
- `HQExpiration`. To refer to the expiration time of a message. The value is a long integer.
- `HQDurable`. To refer to whether a message is durable or not. The value is a string with valid values: `DURABLE` or `NON_DURABLE`.
- `HQTimestamp`. The timestamp of when the message was created. The value is a long integer.
- `HQSize`. The size of a message in bytes. The value is an integer.

Any other identifiers used in core filter expressions will be assumed to be properties of the message.

15

Persistence

In this chapter we will describe how persistence works with HornetQ and how to configure it.

HornetQ ships with a high performance journal. Since HornetQ handles its own persistence, rather than relying on a database or other 3rd party persistence engine it is very highly optimised for the specific messaging use cases.

A HornetQ journal is an *append only* journal. It consists of a set of files on disk. Each file is pre-created to a fixed size and initially filled with padding. As operations are performed on the server, e.g. add message, update message, delete message, records are appended to the journal. When one journal file is full we move to the next one.

Because records are only appended, i.e. added to the end of the journal we minimise disk head movement, i.e. we minimise random access operations which is typically the slowest operation on a disk.

Making the file size configurable means that an optimal size can be chosen, i.e. making each file fit on a disk cylinder. Modern disk topologies are complex and we are not in control over which cylinder(s) the file is mapped onto so this is not an exact science. But by minimising the number of disk cylinders the file is using, we can minimise the amount of disk head movement, since an entire disk cylinder is accessible simply by the disk rotating - the head does not have to move.

As delete records are added to the journal, HornetQ has a sophisticated file garbage collection algorithm which can determine if a particular journal file is needed any more - i.e. has all it's data been deleted in the same or other files. If so, the file can be reclaimed and re-used.

HornetQ also has a compaction algorithm which removes dead space from the journal and compresses up the data so it takes up less files on disk.

The journal also fully supports transactional operation if required, supporting both local and XA transactions.

The majority of the journal is written in Java, however we abstract out the interaction with the actual file system to allow different pluggable implementations. HornetQ ships with two implementations:

- Java NIO [http://en.wikipedia.org/wiki/New_I/O].

  The first implementation uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform where there's a Java 5+ runtime.

- Linux Asynchronous IO

  The second implementation uses a thin native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, HornetQ will be called back when the data has made it to disk, allowing us to avoid explicit syncs altogether and simply send back confirmation of completion when AIO informs us that the data has been persisted.

Using AIO will typically provide even better performance than using Java NIO.

The AIO journal is only available when running Linux kernel 2.6 or later and after having installed libaio (if it's not already installed). For instructions on how to install libaio please see Section 15.4.

For more information on libaio please see Chapter 40.

libaio is part of the kernel project.

The standard HornetQ core server uses two instances of the journal:

- Bindings journal.

  This journal is used to store bindings related data. That includes the set of queues that are deployed on the server and their attributes. It also stores data such as id sequence counters.

  The bindings journal is always a NIO journal as it is typically low throughput compared to the message journal.

- Message journal.

  This journal instance stores all message related data, including the message themselves and also duplicate-id caches.

  By default HornetQ will try and use an AIO journal. If AIO is not available, e.g. the platform is not Linux with the correct kernel version or AIO has not been installed then it will automatically fall back to using Java NIO which is available on any Java platform.

For large messages, HornetQ persists them outside the message journal. This is discussed in Chapter 23.

HornetQ can also be configured to page messages to disk in low memory situations. This is discussed in Chapter 24.

If no persistence is required at all, HornetQ can also be configured not to persist any data at all to storage as discussed in Section 15.5.

15.1. Configuring the bindings journal

The bindings journal is configured using the following attributes in `hornetq-configuration.xml`

- `bindings-directory`

  This is the directory in which the bindings journal lives. The default value is `data/bindings`.

- `create-bindings-dir`

  If this is set to `true` then the bindings directory will be automatically created at the location specified in `bindings-directory` if it does not already exist. The default value is `true`

15.2. Configuring the message journal

The message journal is configured using the following attributes in `hornetq-configuration.xml`

- `journal-directory`

  This is the directory in which the message journal lives. The default value is `data/journal`.

  For the best performance, we recommend the journal is located on its own physical volume in order to minimise disk head movement. If the journal is on a volume which is shared with other processes which might be writing other files (e.g. bindings journal, database, or transaction coordinator) then the disk head may well be moving rapidly between these files as it writes them, thus drastically reducing performance.

  When the message journal is stored on a SAN we recommend each journal instance that is stored on the SAN is given its own LUN (logical unit).

- `create-journal-dir`

  If this is set to `true` then the journal directory will be automatically created at the location specified in `journal-directory` if it does not already exist. The default value is `true`

- `journal-type`

  Valid values are `NIO` or `ASYNCIO`.

  Choosing `NIO` chooses the Java NIO journal. Choosing `AIO` chooses the Linux asynchronous IO journal. If you choose `AIO` but are not running Linux or you do not have libaio installed then HornetQ will detect this and automatically fall back to using `NIO`.

- `journal-sync-transactional`

  If this is set to true then HornetQ will make sure all transaction data is flushed to disk on transaction boundaries (commit, prepare and rollback). The default value is `true`.

- `journal-sync-non-transactional`

  If this is set to true then HornetQ will make sure non transactional message data (sends and acknowledgements) are flushed to disk each time. The default value for this is `true`.

- `journal-file-size`

  The size of each journal file in bytes. The default value for this is `10485760` bytes (10MiB).

- `journal-min-files`

  The minimum number of files the journal will maintain. When HornetQ starts and there is no initial message data, HornetQ will pre-create `journal-min-files` number of files.

  Creating journal files and filling them with padding is a fairly expensive operation and we want to minimise doing this at run-time as files get filled. By precreating files, as one is filled the journal can immediately resume with the next one without pausing to create it.

Depending on how much data you expect your queues to contain at steady state you should tune this number of files to match that total amount of data.

- `journal-max-io`

  Write requests are queued up before being submitted to the system for execution. This parameter controls the maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full then writes will block until space is freed up.

  When using NIO, this value should always be equal to `1`

  When using AIO, the default should be `500`.

  The system maintains different defaults for this parameter depening on whether it's NIO or AIO (default for NIO is 1, default for AIO is 500)

  There is a limit and the total max AIO can't be higher than what is configured at the OS level (/proc/sys/fs/aio-max-nr) usually at 65536.

- `journal-buffer-timeout`

  Instead of flushing on every write that requires a flush, we maintain an internal buffer, and flush the entire buffer either when it is full, or when a timeout expires, whichever is sooner. This is used for both NIO and AIO and allows the system to scale better with many concurrent writes that require flushing.

  This parameter controls the timeout at which the buffer will be flushed if it hasn't filled already. AIO can typically cope with a higher flush rate than NIO, so the system maintains different defaults for both NIO and AIO (default for NIO is 3333333 nanoseconds - 300 times per second, default for AIO is 500000 nanoseconds - ie. 2000 times per second).

  ## Note

  By increasing the timeout, you may be able to increase system throughput at the expense of latency, the default parameters are chosen to give a reasonable balance between throughput and latency.

- `journal-buffer-size`

  The size of the timed buffer on AIO. The default value is `490KiB`.

- `journal-compact-min-files`

  The minimal number of files before we can consider compacting the journal. The compacting algorithm won't start until you have at least `journal-compact-min-files`

  The default for this parameter is `10`

- `journal-compact-percentage`

  The threshold to start compacting. When less than this percentage is considered live data, we start compacting. Note also that compacting won't kick in until you have at least `journal-compact-min-files` data files on the journal

The default for this parameter is `30`

15.3. An important note on disabling disk write cache.

Warning

Most disks contain hardware write caches. A write cache can increase the apparent performance of the disk because writes just go into the cache and are then lazily written to the disk later.

This happens irrespective of whether you have executed a fsync() from the operating system or correctly synced data from inside a Java program!

By default many systems ship with disk write cache enabled. This means that even after syncing from the operating system there is no guarantee the data has actually made it to disk, so if a failure occurs, critical data can be lost.

Some more expensive disks have non volatile or battery backed write caches which won't necessarily lose data on event of failure, but you need to test them!

If your disk does not have an expensive non volatile or battery backed cache and it's not part of some kind of redundant array (e.g. RAID), and you value your data integrity you need to make sure disk write cache is disabled.

Be aware that disabling disk write cache can give you a nasty shock performance wise. If you've been used to using disks with write cache enabled in their default setting, unaware that your data integrity could be compromised, then disabling it will give you an idea of how fast your disk can perform when acting really reliably.

On Linux you can inspect and/or change your disk's write cache settings using the tools `hdparm` (for IDE disks) or `sdparm` or `sginfo` (for SDSI/SATA disks)

On Windows you can check / change the setting by right clicking on the disk and clicking properties.

15.4. Installing AIO

The Java NIO journal gives great performance, but If you are running HornetQ using Linux Kernel 2.6 or later, we highly recommend you use the `AIO` journal for the very best persistence performance.

It's not possible to use the AIO journal under other operating systems or earlier versions of the Linux kernel.

If you are running Linux kernel 2.6 or later and don't already have `libaio` installed, you can easily install it using the following steps:

Using yum, (e.g. on Fedora or Red Hat Enterprise Linux):

```
yum install libaio
```

Using aptitude, (e.g. on Ubuntu or Debian system):

```
apt-get install libaio
```

15.5. Configuring HornetQ for Zero Persistence

In some situations, zero persistence is sometimes required for a messaging system. Configuring HornetQ to perform zero persistence is straightforward. Simply set the parameter `persistence-enabled` in `hornetq-configuration.xml` to `false`.

Please note that if you set this parameter to false, then *zero* persistence will occur. That means no bindings data, message data, large message data, duplicate id caches or paging data will be persisted.

16

Configuring the Transport

HornetQ has a fully pluggable and highly flexible transport layer and defines its own Service Provider Interface (SPI) to make plugging in a new transport provider relatively straightforward.

In this chapter we'll describe the concepts required for understanding HornetQ transports and where and how they're configured.

16.1. Understanding Acceptors

One of the most important concepts in HornetQ transports is the *acceptor*. Let's dive straight in and take a look at an acceptor defined in xml in the configuration file `hornetq-configuration.xml`.

```
<acceptors>
    <acceptor name="netty">
        <factory-class>
org.hornetq.integration.transports.netty.NettyAcceptorFactory
        </factory-class>
        <param key="port" value="5446"/>
    </acceptor>
</acceptors>
```

Acceptors are always defined inside an `acceptors` element. There can be one or more acceptors defined in the `acceptors` element. There's no upper limit to the number of acceptors per server.

Each acceptor defines a way in which connections can be made to the HornetQ server.

In the above example we're defining an acceptor that uses Netty [http://jboss.org/netty] to listen for connections at port `5446`.

The `acceptor` element contains a sub-element `factory-class`, this element defines the factory used to create acceptor instances. In this case we're using Netty to listen for connections so we use the Netty implementation of an `AcceptorFactory` to do this. Basically, the `factory-class` element determines which pluggable transport we're going to use to do the actual listening.

The `acceptor` element can also be configured with zero or more `param` sub-elements. Each `param` element defines a key-value pair. These key-value pairs are used to configure the specific transport, the set of valid key-value pairs depends on the specific transport be used and are passed straight through to the underlying transport.

Examples of key-value pairs for a particular transport would be, say, to configure the IP address to bind to, or the port to listen at.

16.2. Understanding Connectors

Whereas acceptors are used on the server to define how we accept connections, connectors are used by a client to define how it connects to a server.

Let's look at a connector defined in our `hornetq-configuration.xml` file:

```
<connectors>
    <connector name="netty">
        <factory-class>
            org.hornetq.integration.transports.netty.NettyConnectorFactory
        </factory-class>
        <param key="port" value="5446"/>
    </connector>
</connectors>
```

Connectors can be defined inside a `connectors` element. There can be one or more connectors defined in the `connectors` element. There's no upper limit to the number of connectors per server.

You make ask yourself, if connectors are used by the *client* to make connections then why are they defined on the *server*? There are a couple of reasons for this:

- Sometimes the server acts as a client itself when it connects to another server, for example when one server is bridged to another, or when a server takes part in a cluster. In this cases the server needs to know how to connect to other servers. That's defined by *connectors*.
- If you're using JMS and the server side JMS service to instantiate JMS ConnectionFactory instances and bind them in JNDI, then when creating the `HornetQConnectionFactory` it needs to know what server that connection factory will create connections to.

  That's defined by the `connector-ref` element in the `hornetq-jms.xml`file on the server side. Let's take a look at a snipped from a `hornetq-jms.xml` file that shows a JMS connection factory that references our netty connector defined in our `hornetq-configuration.xml` file:

```
<connection-factory name="ConnectionFactory">
    <connectors>
       <connector-ref connector-name="netty"/>
    </connectors>
    <entries>
        <entry name="ConnectionFactory"/>
        <entry name="XAConnectionFactory"/>
    </entries>
</connection-factory>
```

16.3. Configuring the transport directly from the client side.

How do we configure a core `ClientSessionFactory` with the information that it needs to connect with a server?

Connectors are also used indirectly when directly configuring a core `ClientSessionFactory` to directly talk to a server. Although in this case there's no need to define such a connector in the server side configuration, instead we just create the parameters and tell the `ClientSessionFactory` which connector factory to use.

Here's an example of creating a `ClientSessionFactory` which will connect directly to the acceptor we defined earlier in this chapter, it uses the standard Netty TCP transport and will try and connect on port 5446 to localhost (default):

```
Map<String, Object> connectionParams = new HashMap<String, Object>();

connectionParams.put(org.hornetq.integration.transports.netty.TransportConstants.PORT_PROP_NAME,
                     5446);

TransportConfiguration transportConfiguration =
    new TransportConfiguration(
    "org.hornetq.integration.transports.netty.NettyConnectorFactory",
    connectionParams);

ClientSessionFactory sessionFactory = HornetQClient.createClientSessionFactory(transportConfiguration);

ClientSession session = sessionFactory.createSession(...);

etc
```

Similarly, if you're using JMS, you can configure the JMS connection factory directly on the client side without having to define a connector on the server side or define a connection factory in `hornetq-jms.xml`:

```
Map<String, Object> connectionParams = new HashMap<String, Object>();

connectionParams.put(org.hornetq.integration.transports.netty.TransportConstants.PORT_PROP_NAME, 5446);

TransportConfiguration transportConfiguration =
    new TransportConfiguration(
    "org.hornetq.integration.transports.netty.NettyConnectorFactory",
    connectionParams);

ConnectionFactory connectionFactory = HornetQJMSClient.createConnectionFactory(transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();

etc
```

16.4. Configuring the Netty transport

Out of the box, HornetQ currently uses Netty [http://www.jboss.org/netty/], a high performance low level network library.

Our Netty transport can be configured in several different ways; to use old (blocking) Java IO, or NIO (non-blocking), also to use straightforward TCP sockets, SSL, or to tunnel over HTTP or HTTPS, on top of that we also provide a servlet transport.

We believe this caters for the vast majority of transport requirements.

16.4.1. Configuring Netty TCP

Netty TCP is a simple unencrypted TCP sockets based transport. Netty TCP can be configured to use old blocking Java IO or non blocking Java NIO. We recommend you use the Java NIO on the server side for better scalability with many concurrent connections. However using Java old IO can sometimes give you better latency than NIO when you're not so worried about supporting many thousands of concurrent connections.

If you're running connections across an untrusted network please bear in mind this transport is unencrypted. You may want to look at the SSL or HTTPS configurations.

With the Netty TCP transport all connections are initiated from the client side. I.e. the server does not initiate any connections to the client. This works well with firewall policies that typically only allow connections to be initiated in one direction.

All the valid Netty transport keys are defined in the class `org.hornetq.integration.transports.netty.TransportConstants`. The parameters can be used either with acceptors or connectors. The following parameters can be used to configure Netty for simple TCP:

- `use-nio`. If this is `true` then Java non blocking NIO will be used. If set to `false` than old blocking Java IO will be used.

  We highly recommend that you use non blocking Java NIO. Java NIO does not maintain a thread per connection so can scale to many more concurrent connections than with old blocking IO. We recommend the usage of Java 6 for NIO and the best scalability. The default value for this property is `true` on the server side and `false` on the client side.

- `host`. This specifies the host name or IP address to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is `localhost`. When configuring acceptors, multiple hosts or IP addresses can be specified by separating them with commas. It is also possible to specify `0.0.0.0` to accept connection from all the host's network interfaces. It's not valid to specify multiple addresses when specifying the host for a connector; a connector makes a connection to one specific address.

  Note

  Don't forget to specify a host name or ip address! If you want your server able to accept connections from other nodes you must specify a hostname or ip address at which the acceptor will bind and listen for incoming connections. The default is localhost which of course is not accessible from remote nodes!

- `port`. This specified the port to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is `5445`.

- `tcp-no-delay`. If this is `true` then Nagle's algorithm [http://en.wikipedia.org/wiki/Nagle's_algorithm] will be enabled. The default value for this property is `true`.

- `tcp-send-buffer-size`. This parameter determines the size of the TCP send buffer in bytes. The default value for this property is `32768` bytes (32KiB).

  TCP buffer sizes should be tuned according to the bandwidth and latency of your network. Here's a good link that explains the theory behind this [http://www-didc.lbl.gov/TCP-tuning/].

In summary TCP send/receive buffer sizes should be calculated as:

```
buffer_size = bandwidth * RTT.
```

Where bandwidth is in *bytes per second* and network round trip time (RTT) is in seconds. RTT can be easily measured using the `ping` utility.

For fast networks you may want to increase the buffer sizes from the defaults.

- `tcp-receive-buffer-size`. This parameter determines the size of the TCP receive buffer in bytes. The default value for this property is `32768` bytes (32KiB).

16.4.2. Configuring Netty SSL

Netty SSL is similar to the Netty TCP transport but it provides additional security by encrypting TCP connections using the Secure Sockets Layer SSL

Please see the examples for a full working example of using Netty SSL.

Netty SSL uses all the same properties as Netty TCP but adds the following additional properties:

- `ssl-enabled`. Must be `true` to enable SSL.
- `key-store-path`. This is the path to the SSL key store on the client which holds the client certificates.
- `key-store-password`. This is the password for the client certificate key store on the client.
- `trust-store-path`. This is the path to the trusted client certificate store on the server.
- `trust-store-password`. This is the password to the trusted client certificate store on the server.

16.4.3. Configuring Netty HTTP

Netty HTTP tunnels packets over the HTTP protocol. It can be useful in scenarios where firewalls only allow HTTP traffice to pass.

Please see the examples for a full working example of using Netty HTTP.

Netty HTTP uses the same properties as Netty TCP but adds the following additional properties:

- `http-enabled`. Must be `true` to enable HTTP.
- `http-client-idle-time`. How long a client can be idle before sending an empty http request to keep the connection alive
- `http-client-idle-scan-period`. How often, in milliseconds, to scan for idle clients

- `http-response-time`. How long the server can wait before sending an empty http response to keep the connection alive
- `http-server-scan-period`. How often, in milliseconds, to scan for clients needing responses
- `http-requires-session-id`. If true the client will wait after the first call to receive a session id. Used the http connector is connecting to servlet acceptor (not recommended)

16.4.4. Configuring Netty Servlet

We also provide a Netty servlet transport for use with HornetQ. The servlet transport allows HornetQ traffic to be tunneled over HTTP to a servlet running in a servlet engine which then redirects it to an in-VM HornetQ server.

The servlet transport differs from the Netty HTTP transport in that, with the HTTP transport HornetQ effectively acts a web server listening for HTTP traffic on, e.g. port 80 or 8080, whereas with the servlet transport HornetQ traffic is proxied through a servlet engine which may already be serving web site or other applications. This allows HornetQ to be used where corporate policies may only allow a single web server listening on an HTTP port, and this needs to serve all applications including messaging.

Please see the examples for a full working example of the servlet transport being used.

To configure a servlet engine to work the Netty Servlet transport we need to do the following things:

- Deploy the servlet. Here's an example web.xml describing a web application that uses the servlet:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.
         version="2.4">
   <servlet>
      <servlet-name>HornetQServlet</servlet-name>
      <servlet-class>org.jboss.netty.channel.socket.http.HttpTunnelingServlet</servlet-class>
      <init-param>
             <param-name>endpoint</param-name>
             <param-value>local:org.hornetq</param-value>
           </init-param>
           <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HornetQServlet</servlet-name>
      <url-pattern>/HornetQServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

- We also need to add a special Netty invm acceptor on the server side configuration.

  Here's a snippet from the `hornetq-configuration.xml` file showing that acceptor being defined:

```
<acceptors>

      <acceptor name="netty-invm">
         <factory-class>
            org.hornetq.integration.transports.netty.NettyAcceptorFactory
```

```
         </factory-class>
         <param key="use-invm" value="true"/>
         <param key="host" value="org.hornetq"/>
      </acceptor>

</acceptors>
```

- Lastly we need a connector for the client, this again will be configured in the `hornetq-configuration.xml` file as such:

```
<connectors>

      <connector name="netty-servlet">
         <factory-class>
            org.hornetq.integration.transports.netty.NettyConnectorFactory
         </factory-class>
         <param key="host" value="localhost"/>
         <param key="port" value="8080"/>
         <param key="use-servlet" value="true"/>
         <param key="servlet-path" value="/messaging/HornetQServlet"/>
      </connector>

 </connectors>
```

Heres a list of the init params and what they are used for

- endpoint - This is the name of the netty acceptor that the servlet will forward its packets too. You can see it matches the name of the `host` param.

The servlet pattern configured in the `web.xml` is the path of the URL that is used. The connector param `servlet-path` on the connector config must match this using the application context of the web app if there is one.

Its also possible to use the servlet transport over SSL. simply add the following configuration to the connector:

```
    <connector name="netty-servlet">
         <factory-class>org.hornetq.integration.transports.netty.NettyConnectorFactory</factory-class>
         <param key="host" value="localhost"/>
         <param key="port" value="8443"/>
         <param key="use-servlet" value="true"/>
         <param key="servlet-path" value="/messaging/HornetQServlet"/>
         <param key="ssl-enabled" value="true"/>
         <param key="key-store-path" value="path to a keystoree"/>
         <param key="key-store-password" value="keystore password"/>
      </connector>
```

You will also have to configure the Application server to use a KeyStore. Edit the `server.xml` file that can be found under `server/default/deploy/jbossweb.sar` of the Application Server installation and edit the SSL/TLS connector configuration to look like the following:

```
<Connector protocol="HTTP/1.1" SSLEnabled="true"
           port="8443" address="${jboss.bind.address}"
           scheme="https" secure="true" clientAuth="false"
           keystoreFile="path to a keystore"
           keystorePass="keystore password" sslProtocol = "TLS" />
```

In both cases you will need to provide a keystore and password. Take a look at the servlet ssl example shipped with HornetQ for more detail.

17

Detecting Dead Connections

In this section we will discuss connection time-to-live (TTL) and explain how HornetQ deals with crashed clients and clients which have exited without cleanly closing their resources.

17.1. Cleaning up Dead Connection Resources on the Server

Before a HornetQ client application exits it is considered good practice that it should close its resources in a controlled manner, using a `finally` block.

Here's an example of a well behaved core client application closing its session and session factory in a finally block:

```
ClientSessionFactory sf = null;
ClientSession session = null;

try
{
   sf = HornetQClient.createClientSessionFactory(...);

   session = sf.createSession(...);

   ... do some stuff with the session...
}
finally
{
   if (session != null)
   {
      session.close();
   }

   if (sf != null)
   {
      sf.close();
   }
}
```

And here's an example of a well behaved JMS client application:

```
Connection jmsConnection = null;

try
{
   ConnectionFactory jmsConnectionFactory = HornetQJMSClient.createConnectionFactory(...);

   jmsConnection = jmsConnectionFactory.createConnection();
```

```
   ... do some stuff with the connection...
}
finally
{
   if (connection != null)
   {
      connection.close();
   }
}
```

Unfortunately users don't always write well behaved applications, and sometimes clients just crash so they don't have a chance to clean up their resources!

If this occurs then it can leave server side resources, like sessions, hanging on the server. If these were not removed they would cause a resource leak on the server and over time this result in the server running out of memory or other resources.

We have to balance the requirement for cleaning up dead client resources with the fact that sometimes the network between the client and the server can fail and then come back, allowing the client to reconnect. HornetQ supports client reconnection, so we don't want to clean up "dead" server side resources too soon or this will prevent any client from reconnecting, as it won't be able to find its old sessions on the server.

HornetQ makes all of this configurable. For each `ClientSessionFactory` we define a *connection TTL*. Basically, the TTL determines how long the server will keep a connection alive in the absence of any data arriving from the client. The client will automatically send "ping" packets periodically to prevent the server from closing it down. If the server doesn't receive any packets on a connection for the connection TTL time, then it will automatically close all the sessions on the server that relate to that connection.

If you're using JMS, the connection TTL is defined by the `ConnectionTTL` attribute on a `HornetQConnectionFactory` instance, or if you're deploying JMS connection factory instances direct into JNDI on the server side, you can specify it in the xml config, using the parameter `connection-ttl`.

The default value for connection ttl is `60000`ms, i.e. 1 minute. A value of `-1` for `ConnectionTTL` means the server will never time out the connection on the server side.

If you do not wish clients to be able to specify their own connection TTL, you can override all values used by a global value set on the server side. This can be done by specifying the `connection-ttl-override` attribute in the server side configuration. The default value for `connection-ttl-override` is `-1` which means "do not override" (i.e. let clients use their own values).

17.1.1. Closing core sessions or JMS connections that you have failed to close

As previously discussed, it's important that all core client sessions and JMS connections are always closed explicitly in a `finally` block when you are finished using them.

If you fail to do so, HornetQ will detect this at garbage collection time, and log a warning similar to the following in the logs (If you are using JMS the warning will involve a JMS connection not a client session):

```

```

```
[Finalizer] 20:14:43,244 WARNING [org.hornetq.core.client.impl.DelegatingSession]  I'm closin
g a ClientSession you left open. Please make sure you close all ClientSessions explicitly before let
ting them go out of scope!
[Finalizer] 20:14:43,244 WARNING [org.hornetq.core.client.impl.DelegatingSession]  The sessi
on you didn't close was created here:
java.lang.Exception
at org.hornetq.core.client.impl.DelegatingSession.<init>(DelegatingSession.java:83)
at org.acme.yourproject.YourClass (YourClass.java:666)
```

HornetQ will then close the connection / client session for you.

Note that the log will also tell you the exact line of your user code where you created the JMS connection / client session that you later did not close. This will enable you to pinpoint the error in your code and correct it appropriately.

17.2. Detecting failure from the client side.

In the previous section we discussed how the client sends pings to the server and how "dead" connection resources are cleaned up by the server. There's also another reason for pinging, and that's for the *client* to be able to detect that the server or network has failed.

As long as the client is receiving data from the server it will consider the connection to be still alive.

If the client does not receive any packets for `client-failure-check-period` milliseconds then it will consider the connection failed and will either initiate failover, or call any `FailureListener` instances (or `ExceptionListener` instances if you are using JMS) depending on how it has been configured.

If you're using JMS it's defined by the `ClientFailureCheckPeriod` attribute on a `HornetQConnectionFactory` instance, or if you're deploying JMS connection factory instances direct into JNDI on the server side, you can specify it in the `hornetq-jms.xml` configuration file, using the parameter `client-failure-check-period`.

The default value for client failure check period is `30000`ms, i.e. 30 seconds. A value of `-1` means the client will never fail the connection on the client side if no data is received from the server. Typically this is much lower than connection TTL to allow clients to reconnect in case of transitory failure.

17.3. Configuring Asynchronous Connection Execution

By default, packets received on the server side are executed on the remoting thread.

It is possible instead to use a thread from a thread pool to handle some packets so that the remoting thread is not tied up for too long. However, please note that processing operations asynchronously on another thread adds a little more latency. Please note that most short running operations are always handled on the remoting thread for performance reasons. To enable asynchronous connection execution, set the parameter `async-connection-execution-enabled` in `hornetq-configuration.xml` to `true` (default value is `true`).

18

Resource Manager Configuration

HornetQ has its own Resource Manager for handling the lifespan of JTA transactions. When a transaction is started the resource manager is notified and keeps a record of the transaction and its current state. It is possible in some cases for a transaction to be started but then forgotten about. Maybe the client died and never came back. If this happens then the transaction will just sit there indefinitely.

To cope with this HornetQ can, if configured, scan for old transactions and rollback any it finds. The default for this is 3000000 milliseconds (5 minutes), i.e. any transactions older than 5 minutes are removed. This timeout can be changed by editing the `transaction-timeout` property in `hornetq-configuration.xml` (value must be in milliseconds). The property `transaction-timeout-scan-period` configures how often, in milliseconds, to scan for old transactions.

Please note that HornetQ will not unilaterally rollback any XA transactions in a prepared state - this must be heuristically rolled back via the management API if you are sure they will never be resolved by the transaction manager.

19

Flow Control

Flow control is used to limit the flow of data between a client and server, or a server and another server in order to prevent the client or server being overwhelmed with data.

19.1. Consumer Flow Control

This controls the flow of data between the server and the client as the client consumes messages. For performance reasons clients normally buffer messages before delivering to the consumer via the `receive()` method or asynchronously via a message listener. If the consumer cannot process messages as fast as they are being delivered and stored in the internal buffer, then you could end up with a situation where messages would keep building up possibly causing out of memory on the client if they cannot be processed in time.

19.1.1. Window-Based Flow Control

By default, HornetQ consumers buffer messages from the server in a client side buffer before the client consumes them. This improves performance: otherwise every time the client consumes a message, HornetQ would have to go the server to request the next message. In turn, this message would then get sent to the client side, if one was available.

A network round trip would be involved for *every* message and considerably reduce performance.

To prevent this, HornetQ pre-fetches messages into a buffer on each consumer. The total maximum size of messages (in bytes) that will be buffered on each consumer is determined by the `consumer-window-size` parameter.

By default, the `consumer-window-size` is set to 1 MiB (1024 * 1024 bytes).

The value can be:

- `-1` for an *unbounded* buffer
- `0` to not buffer any messages. See Section 11.1.32 for working example of a consumer with no buffering.
- `>0` for a buffer with the given maximum size in bytes.

Setting the consumer window size can considerably improve performance depending on the messaging use case. As an example, let's consider the two extremes:

Fast consumers
: Fast consumers can process messages as fast as they consume them (or even faster)

To allow fast consumers, set the `consumer-window-size` to -1. This will allow *unbounded* message buffering on the client side.

Use this setting with caution: it can overflow the client memory if the consumer is not able to process messages as fast as it receives them.

Slow consumers

Slow consumers takes significant time to process each message and it is desirable to prevent buffering messages on the client side so that they can be delivered to another consumer instead.

Consider a situation where a queue has 2 consumers; 1 of which is very slow. Messages are delivered in a round robin fashion to both consumers, the fast consumer processes all of its messages very quickly until its buffer is empty. At this point there are still messages awaiting to be processed in the buffer of the slow consumer thus preventing them being processed by the fast consumer. The fast consumer is therefore sitting idle when it could be processing the other messages.

To allow slow consumers, set the `consumer-window-size` to 0 (for no buffer at all). This will prevent the slow consumer from buffering any messages on the client side. Messages will remain on the server side ready to be consumed by other consumers.

Setting this to 0 can give deterministic distribution between multiple consumers on a queue.

Most of the consumers cannot be clearly identified as fast or slow consumers but are in-between. In that case, setting the value of `consumer-window-size` to optimize performance depends on the messaging use case and requires benchmarks to find the optimal value, but a value of 1MiB is fine in most cases.

19.1.1.1. Using Core API

If HornetQ Core API is used, the consumer window size is specified by `ClientSessionFactory.setConsumerWindowSize()` method and some of the `ClientSession.createConsumer()` methods.

19.1.1.2. Using JMS

if JNDI is used to look up the connection factory, the consumer window size is configured in `hornetq-jms.xml`:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>

   <!-- Set the consumer window size to 0 to have *no* buffer on the client side -->
   <consumer-window-size>0</consumer-window-size>
</connection-factory>
```

If the connection factory is directly instantiated, the consumer window size is specified by `HornetQConnectionFactory.setConsumerWindowSize()` method.

Please see Section 11.1.32 for an example which shows how to configure HornetQ to prevent consumer buffering

when dealing with slow consumers.

19.1.2. Rate limited flow control

It is also possible to control the *rate* at which a consumer can consume messages. This is a form of throttling and can be used to make sure that a consumer never consumes messages at a rate faster than the rate specified.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to `-1` disables rate limited flow control. The default value is `-1`.

Please see Section 11.1.10 for a working example of limiting consumer rate.

19.1.2.1. Using Core API

If the HornetQ core API is being used the rate can be set via the `ClientSessionFactory.setConsumerMaxRate(int consumerMaxRate)` method or alternatively via some of the `ClientSession.createConsumer()` methods.

19.1.2.2. Using JMS

If JNDI is used to look up the connection factory, the max rate can be configured in `hornetq-jms.xml`:

```
<connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty-connector"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
      <!-- We limit consumers created on this connection factory to consume messages
             at a maximum rate
      of 10 messages per sec -->
      <consumer-max-rate>10</consumer-max-rate>
 </connection-factory>
```

If the connection factory is directly instantiated, the max rate size can be set via the `HornetQConnectionFactory.setConsumerMaxRate(int consumerMaxRate)` method.

> **Note**
>
> Rate limited flow control can be used in conjunction with window based flow control. Rate limited flow control only effects how many messages a client can consume in a second and not how many messages are in its buffer. So if you had a slow rate limit and a high window based limit the clients internal buffer would soon fill up with messages.

Please see Section 11.1.10 for an example which shows how to configure HornetQ to prevent consumer buffering when dealing with slow consumers.

19.2. Producer flow control

HornetQ also can limit the amount of data sent from a client to a server to prevent the server being overwhelmed.

19.2.1. Window based flow control

In a similar way to consumer window based flow control, HornetQ producers, by default, can only send messages to an address as long as they have sufficient credits to do so. The amount of credits required to send a message is given by the size of the message.

As producers run low on credits they request more from the server, when the server sends them more credits they can send more messages.

The amount of credits a producer requests in one go is known as the *window size*.

The window size therefore determines the amount of bytes that can be in-flight at any one time before more need to be requested - this prevents the remoting connection from getting overloaded.

19.2.1.1. Using Core API

If the HornetQ core API is being used, window size can be set via the `ClientSessionFactory.setProducerWindowSize(int producerWindowSize)` method.

19.2.1.2. Using JMS

If JNDI is used to look up the connection factory, the producer window size can be configured in `hornetq-jms.xml`:

```
            <connection-factory name="ConnectionFactory">
               <connectors>
                  <connector-ref connector-name="netty-connector"/>
               </connectors>
               <entries>
                  <entry name="ConnectionFactory"/>
               </entries>
               <producer-window-size>10</producer-window-size>
            </connection-factory>
```

If the connection factory is directly instantiated, the producer window size can be set via the `HornetQConnectionFactory.setProducerWindowSize(int producerWindowSize)` method.

19.2.1.3. Blocking producer window based flow control

Normally the server will always give the same number of credits as have been requested. However, it is also possible to set a maximum size on any address, and the server will never send more credits than could cause the address's upper memory limit to be exceeded.

For example, if I have a JMS queue called "myqueue", I could set the maximum memory size to 10MiB, and the the server will control the number of credits sent to any producers which are sending any messages to myqueue such that the total messages in the queue never exceeds 10MiB.

When the address gets full, producers will block on the client side until more space frees up on the address, i.e. until messages are consumed from the queue thus freeing up space for more messages to be sent.

We call this blocking producer flow control, and it's an efficient way to prevent the server running out of memory due to producers sending more messages than can be handled at any time.

It is an alternative approach to paging, which does not block producers but instead pages messages to storage.

To configure an address with a maximum size and tell the server that you want to block producers for this address if it becomes full, you need to define an AddressSettings (Section 25.3) block for the address and specify `max-size-bytes` and `address-full-policy`

The address block applies to all queues registered to that address. I.e. the total memory for all queues bound to that address will not exceed `max-size-bytes`. In the case of JMS topics this means the *total* memory of all subscriptions in the topic won't exceed max-size-bytes.

Here's an example:

```
<address-settings>
   <address-setting match="jms.queue.exampleQueue">
      <max-size-bytes>100000</max-size-bytes>
      <address-full-policy>DROP</address-full-policy>
   </address-setting>
</address-settings>
```

The above example would set the max size of the JMS queue "exampleQueue" to be 100000 bytes and would block any producers sending to that address to prevent that max size being exceeded.

Note the policy must be set to `DROP` to enable blocking producer flow control.

Please note the default value for `address-full-policy` is to `PAGE`. Please see the chapter on paging for more information on paging.

19.2.2. Rate limited flow control

HornetQ also allows the rate a producer can emit message to be limited, in units of messages per second. By specifying such a rate, HornetQ will ensure that producer never produces messages at a rate higher than that specified.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to `-1` disables rate limited flow control. The default value is `-1`.

Please see the Section 11.1.36 for a working example of limiting producer rate.

19.2.2.1. Using Core API

If the HornetQ core API is being used the rate can be set via the `ClientSessionFactory.setProducerMaxRate(int consumerMaxRate)` method or alternatively via some of the `ClientSession.createProducer()` methods.

19.2.2.2. Using JMS

If JNDI is used to look up the connection factory, the max rate can be configured in `hornetq-jms.xml`:

```
<connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty-connector"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
      <!-- We limit producers created on this connection factory to produce messages
                at a maximum rate
      of 10 messages per sec -->
      <producer-max-rate>10</producer-max-rate>
 </connection-factory>
```

If the connection factory is directly instantiated, the max rate size can be set via the `HornetQConnectionFactory.setProducerMaxRate(int consumerMaxRate)` method.

20

Guarantees of sends and commits

20.1. Guarantees of Transaction Completion

When committing or rolling back a transaction with HornetQ, the request to commit or rollback is sent to the server, and the call will block on the client side until a response has been received from the server that the commit or rollback was executed.

When the commit or rollback is received on the server, it will be committed to the journal, and depending on the value of the parameter `journal-sync-transactional` the server will ensure that the commit or rollback is durably persisted to storage before sending the response back to the client. If this parameter has the value `false` then commit or rollback may not actually get persisted to storage until some time after the response has been sent to the client. In event of server failure this may mean the commit or rollback never gets persisted to storage. The default value of this parameter is `true` so the client can be sure all transaction commits or rollbacks have been persisted to storage by the time the call to commit or rollback returns.

Setting this parameter to `false` can improve performance at the expense of some loss of transaction durability.

This parameter is set in `hornetq-configuration.xml`

20.2. Guarantees of Non Transactional Message Sends

If you are sending messages to a server using a non transacted session, HornetQ can be configured to block the call to send until the message has definitely reached the server, and a response has been sent back to the client. This can be configured individually for durable and non-durable messages, and is determined by the following two parameters:

- `BlockOnDurableSend`. If this is set to `true` then all calls to send for durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `true`.
- `BlockOnNonDurableSend`. If this is set to `true` then all calls to send for non-durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `false`.

Setting block on sends to `true` can reduce performance since each send requires a network round trip before the next send can be performed. This means the performance of sending messages will be limited by the network round trip time (RTT) of your network, rather than the bandwidth of your network. For better performance we recommend either batching many messages sends together in a transaction since with a transactional session, only the commit / rollback blocks not every send, or, using HornetQ's advanced *asynchronous send acknowledgements fea-*

ture described in Section 20.4.

If you are using JMS and you're using the JMS service on the server to load your JMS connection factory instances into JNDI then these parameters can be configured in `hornetq-jms.xml` using the elements `block-on-durable-send` and `block-on-non-durable-send`. If you're using JMS but not using JNDI then you can set these values directly on the `HornetQConnectionFactory` instance using the appropriate setter methods.

If you're using core you can set these values directly on the `ClientSessionFactory` instance using the appropriate setter methods.

When the server receives a message sent from a non transactional session, and that message is durable and the message is routed to at least one durable queue, then the server will persist the message in permanent storage. If the journal parameter `journal-sync-non-transactional` is set to `true` the server will not send a response back to the client until the message has been persisted and the server has a guarantee that the data has been persisted to disk. The default value for this parameter is `true`.

20.3. Guarantees of Non Transactional Acknowledgements

If you are acknowledging the delivery of a message at the client side using a non transacted session, HornetQ can be configured to block the call to acknowledge until the acknowledge has definitely reached the server, and a response has been sent back to the client. This is configured with the parameter `BlockOnAcknowledge`. If this is set to `true` then all calls to acknowledge on non transacted sessions will block until the acknowledge has reached the server, and a response has been sent back. You might want to set this to `true` if you want to implement a strict *at most once* delivery policy. The default value is `false`

20.4. Asynchronous Send Acknowledgements

If you are using a non transacted session but want a guarantee that every message sent to the server has reached it, then, as discussed in Section 20.2, you can configure HornetQ to block the call to send until the server has received the message, persisted it and sent back a response. This works well but has a severe performance penalty - each call to send needs to block for at least the time of a network round trip (RTT) - the performance of sending is thus limited by the latency of the network, *not* limited by the network bandwidth.

Let's do a little bit of maths to see how severe that is. We'll consider a standard 1Gib ethernet network with a network round trip between the server and the client of 0.25 ms.

With a RTT of 0.25 ms, the client can send *at most* 1000/ 0.25 = 4000 messages per second if it blocks on each message send.

If each message is < 1500 bytes and a standard 1500 bytes MTU size is used on the network, then a 1GiB network has a *theoretical* upper limit of (1024 * 1024 * 1024 / 8) / 1500 = 89478 messages per second if messages are sent without blocking! These figures aren't an exact science but you can clearly see that being limited by network RTT can have serious effect on performance.

To remedy this, HornetQ provides an advanced new feature called *asynchronous send acknowledgements*. With this feature, HornetQ can be configured to send messages without blocking in one direction and asynchronously getting acknowledgement from the server that the messages were received in a separate stream. By de-coupling the send from the acknowledgement of the send, the system is not limited by the network RTT, but is limited by the

network bandwidth. Consequently better throughput can be achieved than is possible using a blocking approach, while at the same time having absolute guarantees that messages have successfully reached the server.

The window size for send acknowledgements is determined by the confirmation-window-size parameter on the connection factory or client session factory. Please see Chapter 34 for more info on this.

20.4.1. Asynchronous Send Acknowledgements

To use the feature using the core API, you implement the interface `org.hornetq.api.core.client.SendAcknowledgementHandler` and set a handler instance on your `ClientSession`.

Then, you just send messages as normal using your `ClientSession`, and as messages reach the server, the server will send back an acknowledgement of the send asynchronously, and some time later you are informed at the client side by HornetQ calling your handler's `sendAcknowledged(ClientMessage message)` method, passing in a reference to the message that was sent.

To enable asynchronous send acknowledgements you must make sure `confirmation-window-size` is set to a positive integer value, e.g. 10MiB

Please see Section 11.1.45 for a full working example.

21

Message Redelivery and Undelivered Messages

Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back). Such a message goes back to its queue ready to be redelivered. However, this means it is possible for a message to be delivered again and again without any success and remain in the queue, clogging the system.

There are 2 ways to deal with these undelivered messages:

- Delayed redelivery.

  It is possible to delay messages redelivery to let the client some time to recover from transient failures and not overload its network or CPU resources

- Dead Letter Address.

  It is also possible to configure a dead letter address so that after a specified number of unsuccessful deliveries, messages are removed from the queue and will not be delivered again

Both options can be combined for maximum flexibility.

21.1. Delayed Redelivery

Delaying redelivery can often be useful in the case that clients regularly fail or rollback. Without a delayed redelivery, the system can get into a "thrashing" state, with delivery being attempted, the client rolling back, and delivery being re-attempted ad infinitum in quick succession, consuming valuable CPU and network resources.

21.1.1. Configuring Delayed Redelivery

Delayed redelivery is defined in the address-setting configuration:

```
<!-- delay redelivery of messages for 5s -->
<address-setting match="jms.queue.exampleQueue">
   <redelivery-delay>5000</redelivery-delay>
</address-setting>
```

If a `redelivery-delay` is specified, HornetQ will wait this delay before redelivering the messages

By default, there is no redelivery delay (`redelivery-delay`is set to 0).

Address wildcards can be used to configure redelivery delay for a set of addresses (see Chapter 13), so you don't have to specify redelivery delay individually for each address.

21.1.2. Example

See Section 11.1.12 for an example which shows how delayed redelivery is configured and used with JMS.

21.2. Dead Letter Addresses

To prevent a client infinitely receiving the same undelivered message (regardless of what is causing the unsuccessful deliveries), messaging systems define *dead letter addresses*: after a specified unsuccessful delivery attempts, the message is removed from the queue and send instead to a dead letter address.

Any such messages can then be diverted to queue(s) where they can later be perused by the system administrator for action to be taken.

HornetQ's addresses can be assigned a dead letter address. Once the messages have be unsuccessfully delivered for a given number of attempts, they are removed from the queue and sent to the dead letter address. These *dead letter* messages can later be consumed for further inspection.

21.2.1. Configuring Dead Letter Addresses

Dead letter address is defined in the address-setting configuration:

```
<!-- undelivered messages in exampleQueue will be sent to the dead letter address
   deadLetterQueue after 3 unsuccessful delivery attempts
 -->
<address-setting match="jms.queue.exampleQueue">
   <dead-letter-address>jms.queue.deadLetterQueue</dead-letter-address>
   <max-delivery-attempts>3</max-delivery-attempts>
</address-setting>
```

If a `dead-letter-address` is not specified, messages will removed after `max-delivery-attempts` unsuccessful attempts.

By default, messages are redelivered 10 times at the maximum. Set `max-delivery-attempts` to -1 for infinite redeliveries.

For example, a dead letter can be set globally for a set of matching addresses and you can set `max-delivery-attempts` to -1 for a specific address setting to allow infinite redeliveries only for this address.

Address wildcards can be used to configure dead letter settings for a set of addresses (see Chapter 13).

21.2.2. Dead Letter Properties

Dead letter messages which are consumed from a dead letter address have the following property:

- `_HQ_ORIG_ADDRESS`

  a String property containing the *original address* of the dead letter message

21.2.3. Example

See Section 11.1.11 for an example which shows how dead letter is configured and used with JMS.

21.3. Delivery Count Persistence

In normal use, HornetQ does not update delivery count *persistently* until a message is rolled back (i.e. the delivery count is not updated *before* the message is delivered to the consumer). In most messaging use cases, the messages are consumed, acknowledged and forgotten as soon as they are consumed. In these cases, updating the delivery count persistently before delivering the message would add an extra persistent step *for each message delivered*, implying a significant performance penalty.

However, if the delivery count is not updated persistently before the message delivery happens, in the event of a server crash, messages might have been delivered but that will not have been reflected in the delivery count. During the recovery phase, the server will not have knowledge of that and will deliver the message with `redelivered` set to `false` while it should be `true`.

As this behavior breaks strict JMS semantics, HornetQ allows to persist delivery count before message delivery but disabled it by default for performance implications.

To enable it, set `persist-delivery-count-before-delivery` to `true` in `hornetq-configuration.xml`:

```
<persist-delivery-count-before-delivery>true</persist-delivery-count-before-delivery>
```

22

Message Expiry

Messages can be set with an optional *time to live* when sending them.

HornetQ will not deliver a message to a consumer after it's time to live has been exceeded. If the message hasn't been delivered by the time that time to live is reached the server can discard it.

HornetQ's addresses can be assigned a expiry address so that, when messages are expired, they are removed from the queue and sent to the expiry address. Many different queues can be bound to an expiry address. These *expired* messages can later be consumed for further inspection.

22.1. Message Expiry

Using HornetQ Core API, you can set an expiration time directly on the message:

```
// message will expire in 5000ms from now
message.setExpiration(System.currentTimeMillis() + 5000);
```

JMS MessageProducer allows to set a TimeToLive for the messages it sent:

```
// messages sent by this producer will be retained for 5s (5000ms) before expiration
producer.setTimeToLive(5000);
```

Expired messages which are consumed from an expiry address have the following properties:

- `_HQ_ORIG_ADDRESS`

  a String property containing the *original address* of the expired message

- `_HQ_ACTUAL_EXPIRY`

  a Long property containing the *actual expiration time* of the expired message

22.2. Configuring Expiry Addresses

Expiry address are defined in the address-setting configuration:

```
<!-- expired messages in exampleQueue will be sent to the expiry address expiryQueue -->
```

```
<address-setting match="jms.queue.exampleQueue">
   <expiry-address>jms.queue.expiryQueue</expiry-address>
</address-setting>
```

If messages are expired and no expiry address is specified, messages are simply removed from the queue and dropped. Address wildcards can be used to configure expiry address for a set of addresses (see Chapter 13).

22.3. Configuring The Expiry Reaper Thread

A reaper thread will periodically inspect the queues to check if messages have expired.

The reaper thread can be configured with the following properties in `hornetq-configuration.xml`

- `message-expiry-scan-period`

  How often the queues will be scanned to detect expired messages (in milliseconds, default is 30000ms)

- `message-expiry-thread-priority`

  The reaper thread priority (it must be between 0 and 9, 9 being the highest priority, default is 3)

22.4. Example

See Section 11.1.28 for an example which shows how message expiry is configured and used with JMS.

23

Large Messages

HornetQ supports sending and receiving of huge messages, even when the client and server are running with limited memory. The only realistic limit to the size of a message that can be sent or consumed is the amount of disk space you have available. We have tested sending and consuming messages up to 8 GiB in size with a client and server running in just 50MiB of RAM!

To send a large message, the user can set an `InputStream` on a message body, and when that message is sent, HornetQ will read the `InputStream`. A `FileInputStream` could be used for example to send a huge message from a huge file on disk.

As the `InputStream` is read the data is sent to the server as a stream of fragments. The server persists these fragments to disk as it receives them and when the time comes to deliver them to a consumer they are read back of the disk, also in fragments and sent down the wire. When the consumer receives a large message it initially receives just the message with an empty body, it can then set an `OutputStream` on the message to stream the huge message body to a file on disk or elsewhere. At no time is the entire message body stored fully in memory, either on the client or the server.

23.1. Configuring the server

Large messages are stored on a disk directory on the server side, as configured on the main configuration file.

The configuration property `large-messages-directory` specifies where large messages are stored.

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

...

<large-message-directory>/data/large-messages</large-message-directory>

...

</configuration
```

By default the large message directory is `data/largemessages`

For the best performance we recommend large messages directory is stored on a different physical volume to the message journal or paging directory.

23.2. Setting the limits

Any message large than a certain size is considered a large message. Large messages will be split up and sent in fragments. This is determined by the parameter `min-large-message-size`

The default value is 100KiB.

23.2.1. Using Core API

If the HornetQ Core API is used, the minimal large message size is specified by `ClientSessionFactory.setMinLargeMessageSize`.

```
ClientSessionFactory factory =
            HornetQClient.createClientSessionFactory(new
            TransportConfiguration(NettyConnectorFactory.class.getName()), null);
factory.setMinLargeMessageSize(25 * 1024);
```

Section 16.3 will provide more information on how to instantiate the session factory.

23.2.2. Using JMS

If JNDI is used to look up the connection factory, the minimum large message size is specified in `hornetq-jms.xml`

```
...
<connection-factory name="ConnectionFactory">
<connectors>
   <connector-ref connector-name="netty"/>
</connectors>
<entries>
   <entry name="ConnectionFactory"/>
   <entry name="XAConnectionFactory"/>
</entries>

<min-large-message-size>250000</min-large-message-size>
</connection-factory>
...
```

If the connection factory is being instantiated directly, the minimum large message size is specified by `HornetQConnectionFactory.setMinLargeMessageSize`.

23.3. Streaming large messages

HornetQ supports setting the body of messages using input and output streams (`java.lang.io`)

These streams are then used directly for sending (input streams) and receiving (output streams) messages.

When receiving messages there are 2 ways to deal with the output stream; you may choose to block while the output stream is recovered using the method `ClientMessage.saveOutputStream` or alternatively using the method `ClientMessage.setOutputstream` which will asynchronously write the message to the stream. If you choose the latter the consumer must be kept alive until the message has been fully received.

You can use any kind of stream you like. The most common use case is to send files stored in your disk, but you

could also send things like JDBC Blobs, `SocketInputStream`, things you recovered from `HTTPRequests` etc. Anything as long as it implements `java.io.InputStream` for sending messages or `java.io.OutputStream` for receiving them.

23.3.1. Streaming over Core API

The following table shows a list of methods available at `ClientMessage` which are also available through JMS by the use of object properties.

Table 23.1. org.hornetq.api.core.client.ClientMessage API

Name	Description	JMS Equivalent Property
setBodyInputStream(InputStream)	Set the InputStream used to read a message body when sending it.	JMS_HQ_InputStream
setOutputStream(OutputStream)	Set the OutputStream that will receive the body of a message. This method does not block.	JMS_HQ_OutputStream
saveOutputStream(OutputStream)	Save the body of the message to the `OutputStream`. It will block until the entire content is transferred to the `OutputStream`.	JMS_HQ_SaveStream

To set the output stream when receiving a core message:

```
...
ClientMessage msg = consumer.receive(...);


// This will block here until the stream was transferred
msg.saveOutputStream(someOutputStream);

ClientMessage msg2 = consumer.receive(...);

// This will not wait the transfer to finish
msg.setOutputStream(someOtherOutputStream);
...
```

Set the input stream when sending a core message:

```
...
ClientMessage msg = session.createMessage();
msg.setInputStream(dataInputStream);
...
```

23.3.2. Streaming over JMS

When using JMS, HornetQ maps the streaming methods on the core API (see Table 23.1) by setting object properties . You can use the method `Message.setObjectProperty` to set the input and output streams.

The `InputStream` can be defined through the JMS Object Property JMS_HQ_InputStream on messages being sent:

```
BytesMessage message = session.createBytesMessage();

FileInputStream fileInputStream = new FileInputStream(fileInput);

BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);

message.setObjectProperty("JMS_HQ_InputStream", bufferedInput);

someProducer.send(message);
```

The `OutputStream` can be set through the JMS Object Property JMS_HQ_SaveStream on messages being received in a blocking way.

```
BytesMessage messageReceived = (BytesMessage)messageConsumer.receive(120000);

File outputFile = new File("huge_message_received.dat");

FileOutputStream fileOutputStream = new FileOutputStream(outputFile);

BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStream);

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_HQ_SaveStream", bufferedOutput);
```

Setting the `OutputStream` could also be done in a non blocking way using the property JMS_HQ_OutputStream.

```
// This won't wait the stream to finish. You need to keep the consumer active.
messageReceived.setObjectProperty("JMS_HQ_OutputStream", bufferedOutput);
```

Note

When using JMS, Streaming large messages are only supported on `StreamMessage` and `BytesMessage`.

23.4. Streaming Alternative

If you choose not to use the `InputStream` or `OutputStream` capability of HornetQ You could still access the data directly in an alternative fashion.

On the Core API just get the bytes of the body as you normally would.

```
ClientMessage msg = consumer.receive();

byte[] bytes = new byte[1024];
for (int i = 0 ;  i < msg.getBodySize(); i += bytes.length)
{
   msg.getBody().readBytes(bytes);
```

```
   // Whatever you want to do with the bytes
}
```

If using JMS API, `BytesMessage` and `StreamMessage` also supports it transparently.

```
BytesMessage rm = (BytesMessage)cons.receive(10000);

byte data[] = new byte[1024];

for (int i = 0; i < rm.getBodyLength(); i += 1024)
{
   int numberOfBytes = rm.readBytes(data);
   // Do whatever you want with the data
}
```

23.5. Cache Large Messages on client

LargeMessages are transferred by streaming from server to client. The message is broken into smaller packets and as the message is read more packets will be received. Because of that the body of the large message can be read only once, and by consequence a received message can be sent to another producer only once. The JMS Bridge for instance won't be able to resend a large message in case of failure

To solve this problem, you can enable the property `cache-large-message-client` on the connection factory. If you enable this property the client consumer will create a temporary file to hold the large message content, so it would be possible to resend large messages.

Note

Use this option on the connection factory used by the JMS Bridge if the JMS Bridge is being used for large messages.

23.6. Large message example

Please see Section 11.1.22 for an example which shows how large message is configured and used with JMS.

24

Paging

HornetQ transparently supports huge queues containing millions of messages while the server is running with limited memory.

In such a situation it's not possible to store all of the queues in memory at any one time, so HornetQ transparently *pages* messages into and out of memory as they are needed, thus allowing massive queues with a low memory footprint.

HornetQ will start paging messages to disk, when the size of all messages in memory for an address exceeds a configured maximum size.

By default, HornetQ does not page messages - this must be explicitly configured to activate it.

24.1. Page Files

Messages are stored per address on the file system. Each address has an individual folder where messages are stored in multiple files (page files). Each file will contain messages up to a max configured size (`page-size-bytes`). When reading page-files all messages on the page-file are read, routed and the file is deleted as soon as the messages are recovered.

24.2. Configuration

You can configure the location of the paging folder

Global paging parameters are specified on the main configuration file (`hornetq-configuration.xml`).

```
<configuration xmlns="urn:hornetq"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

            ...

            <paging-directory>/somewhere/paging-directory</paging-directory>

            ...
```

Table 24.1. Paging Configuration Parameters

Property Name	Description	Default
`paging-directory`	Where page files are stored. Hor-	data/paging

Property Name	Description	Default
	netQ will create one folder for each address being paged under this configured location.	

24.3. Paging Mode

As soon as messages delivered to an address exceed the configured size, that address alone goes into page mode.

24.3.1. Configuration

Configuration is done at the address settings, done at the main configuration file (`hornetq-configuration.xml`).

```
<address-settings>
    <address-setting match="jms.someaddress">
       <max-size-bytes>104857600</max-size-bytes>
       <page-size-bytes>10485760</page-size-bytes>
       <address-full-policy>PAGE</address-full-policy>
    </address-setting>
 </address-settings>
```

This is the list of available parameters on the address settings.

Table 24.2. Paging Address Settings

Property Name	Description	Default
`max-size-bytes`	What's the max memory the address could have before entering on page mode.	-1 (disabled)
`page-size-bytes`	The size of each page file used on the paging system	10MiB (10 * 1024 * 1024 bytes)
`address-full-message-policy`	This must be set to PAGE for paging to enable. If the value is PAGE then further messages will be paged to disk. If the value is DROP then further messages will be silently dropped. If the value is BLOCK then client message producers will block when they try and send further messages.	PAGE

24.4. Dropping messages

Instead of paging messages when the max size is reached, an address can also be configured to just drop messages when the address is full.

To do this just set the `address-full-policy` to `DROP` in the address settings

24.5. Blocking producers

Instead of paging messages when the max size is reached, an address can also be configured to block producers from sending further messages when the address is full, thus preventing the memory being exhausted on the server.

When memory is freed up on the server, producers will automatically unblock and be able to continue sending.

To do this just set the `address-full-policy` to `BLOCK` in the address settings

24.6. Caution with Addresses with Multiple Queues

When a message is routed to an address that has multiple queues bound to it, e.g. a JMS subscription, there is only 1 copy of the message in memory. Each queue only deals with a reference to this. Because of this the memory is only freed up once all queues referencing the message have delivered it. This means that if not all queues deliver the message we can end up in a state where messages are not delivered.

For example:

- An address has 10 queues
- One of the queues does not deliver its messages (maybe because of a slow consumer).
- Messages continually arrive at the address and paging is started.
- The other 9 queues are empty even though messages have been sent.

In this example we have to wait until the last queue has delivered some of its messages before we depage and the other queues finally receive some more messages.

24.7. Paging and message selectors

Note

Please note that message selectors will only operate on messages in memory. If you have a large amount of messages paged to disk and a selector that only matches some of the paged messages, then those messages won't be consumed until the messages in memory have been consumed. HornetQ does not scan through page files on disk to locate matching messages. To do this efficiently would mean implementing and managing indexes amongst other things. Effectively we would be writing a relational database! This is not the primary role of a messaging system. If you find yourself using selectors which only select small subsets of

messages in very large queues which are too large to fit in memory at any one time, then you probably want a relational database not a messaging system - you're effectively executing queries over tables.

24.8. Paging and browsers

Note

Please note that message browsers only operate over messages in memory. They do not operate over messages paged to disk. Messages are paged to disk *before* they are routed to any queues, so when they are paged, they are not in any queues, so will not appear when browsing any queues.

24.9. Paging and unacknowledged messages

Note

Please note that until messages are acknowledged they are still in memory on the server, so they contribute to the size of messages on a particular address. If messages are paged to disk for an address, and are being consumed, they will be depaged from disk when enough memory has been freed up in that address after messages have been consumed and acknowledged. However if messages are not acknowledged then more messages will not be depaged since there is no free space in memory. In this case message consumption can appear to hang. If not acknowledging explictly messages are acknowledged according to the `ack-batch-size` setting. Be careful not to set your paging max size to a figure lower than ack-batch-size or your system may appear to hang!

24.10. Example

See Section 11.1.34 for an example which shows how to use paging with HornetQ.

25

Queue Attributes

Queue attributes can be set in one of two ways. Either by configuring them using the configuration file or by using the core API. This chapter will explain how to configure each attribute and what effect the attribute has.

25.1. Predefined Queues

Queues can be predefined via configuration at a core level or at a JMS level. Firstly lets look at a JMS level.

The following shows a queue predefined in the `hornetq-jms.xml` configuration file.

```
<queue name="selectorQueue">
      <entry name="/queue/selectorQueue"/>
      <selector string="color='red'"/>
      <durable>true</durable>
</queue>
```

This name attribute of queue defines the name of the queue. When we do this at a jms level we follow a naming convention so the actual name of the core queue will be `jms.queue.selectorQueue`.

The entry element configures the name that will be used to bind the queue to JNDI. This is a mandatory element and the queue can contain multiple of these to bind the same queue to different names.

The selector element defines what JMS message selector the predefined queue will have. Only messages that match the selector will be added to the queue. This is an optional element with a default of null when omitted.

The durable element specifies whether the queue will be persisted. This again is optional and defaults to true if omitted.

Secondly a queue can be predefined at a core level in the `hornetq-configuration.xml` file. The following is an example.

```
<queues>
           <queue name="jms.queue.selectorQueue">
               <address>jms.queue.selectorQueue</address>
               <filter string="color='red'"/>
       <durable>true</durable>
           </queue>
</queues>
```

This is very similar to the JMS configuration, with 3 real differences which are.

1. The name attribute of queue is the actual name used for the queue with no naming convention as in JMS.

2. The address element defines what address is used for routing messages.

3. There is no entry element.

4. The filter uses the *Core filter syntax* (described in Chapter 14), *not* the JMS selector syntax.

25.2. Using the API

Queues can also be created using the core API or the management API.

For the core API, queues can be created via the `org.hornetq.api.core.client.ClientSession` interface. There are multiple `createQueue` methods that support setting all of the previously mentioned attributes. There is one extra attribute that can be set via this API which is `temporary`. setting this to true means that the queue will be deleted once the session is disconnected.

Take a look at Chapter 30 for a description of the management API for creating queues.

25.3. Configuring Queues Via Address Settings

There are some attributes that are defined against an address wildcard rather than a specific queue. Here an example of an `address-setting` entry that would be found in the `hornetq-configuration.xml` file.

```
<address-settings>
    <address-setting match="jms.queue.exampleQueue">
        <dead-letter-address>jms.queue.deadLetterQueue</dead-letter-address>
        <max-delivery-attempts>3</max-delivery-attempts>
        <redelivery-delay>5000</redelivery-delay>
        <expiry-address>jms.queue.expiryQueue</expiry-address>
        <last-value-queue>true</last-value-queue>
        <max-size-bytes>100000</max-size-bytes>
        <page-size-bytes>20000</page-size-bytes>
        <redistribution-delay>0</redistribution-delay>
        <send-to-dla-on-no-route>true</send-to-dla-on-no-route>
        <address-full-policy>PAGE</address-full-policy>
     </address-setting>
</address-settings>
```

The idea with address settings, is you can provide a block of settings which will be applied against any adresses that match the string in the `match` attribute. In the above example the settings would only be applied to any addresses which exactly match the address `jms.queue.exampleQueue`, but you can also use wildcards to apply sets of configuration against many addresses. The wildcard syntax used is described here.

For example, if you used the `match` string `jms.queue.#` the settings would be applied to all addresses which start with `jms.queue.` which would be all JMS queues.

The meaning of the specific settings are explained fully throughout the user manual, however here is a brief description with a link to the appropriate chapter if available.

`max-delivery-attempts` defines how many time a cancelled message can be redelivered before sending to the `dead-letter-address`. A full explanation can be found here.

`redelivery-delay` defines how long to wait before attempting redelivery of a cancelled message. see here.

`expiry-address` defines where to send a message that has expired. see here.

`last-value-queue` defines whether a queue only uses last values or not. see here.

`max-size-bytes` and `page-size-bytes` are used to set paging on an address. This is explained here.

`redistribution-delay` defines how long to wait when the last consumer is closed on a queue before redistributing any messages. see here.

`send-to-dla-on-no-route`. If a message is sent to an address, but the server does not route it to any queues, for example, there might be no queues bound to that address, or none of the queues have filters that match, then normally that message would be discarded. However if this parameter is set to true for that address, if the message is not routed to any queues it will instead be sent to the dead letter address (DLA) for that address, if it exists.

`address-full-policy`. This attribute can have one of the following values: PAGE, DROP or BLOCK and determines what happens when an address where `max-size-bytes` is specified becomes full. The default value is PAGE. If the value is PAGE then further messages will be paged to disk. If the value is DROP then further messages will be silently dropped. If the value is BLOCK then client message producers will block when they try and send further messages. See the following chapters for more info Chapter 19, Chapter 24.

26

Scheduled Messages

Scheduled messages differ from normal messages in that they won't be delivered until a specified time in the future, at the earliest.

To do this, a special property is set on the message before sending it.

26.1. Scheduled Delivery Property

The property name used to identify a scheduled message is `"_HQ_SCHED_DELIVERY"` (or the constant `Message.HDR_SCHEDULED_DELIVERY_TIME`).

The specified value must be a `long` corresponding to the time the message must be delivered (in milliseconds). An example of sending a scheduled message using the JMS API is as follows.

```
TextMessage message =
 session.createTextMessage("This is a scheduled message message which will be delivered
   in 5 sec.");
message.setLongProperty("_HQ_SCHED_DELIVERY", System.currentTimeMillis() + 5000);
producer.send(message);

...

// message will not be received immediately but 5 seconds later
TextMessage messageReceived = (TextMessage) consumer.receive();
```

Scheduled messages can also be sent using the core API, by setting the same property on the core message before sending.

26.2. Example

See Section 11.1.43 for an example which shows how scheduled messages can be used with JMS.

27

Last-Value Queues

Last-Value queues are special queues which discard any messages when a newer message with the same value for a well-defined Last-Value property is put in the queue. In other words, a Last-Value queue only retains the last value.

A typical example for Last-Value queue is for stock prices, where you are only interested by the latest value for a particular stock.

27.1. Configuring Last-Value Queues

Last-value queues are defined in the address-setting configuration:

```
<address-setting match="jms.queue.lastValueQueue">
    <last-value-queue>true</last-value-queue>
</address-setting>
```

By default, `last-value-queue` is false. Address wildcards can be used to configure Last-Value queues for a set of addresses (see Chapter 13).

27.2. Using Last-Value Property

The property name used to identify the last value is `"_HQ_LVQ_NAME"` (or the constant `Message.HDR_LAST_VALUE_NAME` from the Core API).

For example, if two messages with the same value for the Last-Value property are sent to a Last-Value queue, only the latest message will be kept in the queue:

```
// send 1st message with Last-Value property set to STOCK_NAME
TextMessage message =
  session.createTextMessage("1st message with Last-Value property set");
message.setStringProperty("_HQ_LVQ_NAME", "STOCK_NAME");
producer.send(message);

// send 2nd message with Last-Value property set to STOCK_NAME
message =
  session.createTextMessage("2nd message with Last-Value property set");
message.setStringProperty("_HQ_LVQ_NAME", "STOCK_NAME");
producer.send(message);

...

// only the 2nd message will be received: it is the latest with
// the Last-Value property set
```

```
TextMessage messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.format("Received message: %s\n", messageReceived.getText());
```

27.3. Example

See Section 11.1.23 for an example which shows how last value queues are configured and used with JMS.

28

Message Grouping

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group id, i.e. they have same group identifier property (`JMSXGroupID` for JMS, `_HQ_GROUP_ID` for HornetQ Core API).
- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. They pin all messages with the same group id to the same consumer. If that consumer closes another consumer is chosen and will receive all messages with the same group id.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer.

An example might be orders for a certain stock. You may want orders for any particular stock to be processed serially by the same consumer. To do this you can create a pool of consumers (perhaps one for each stock, but less will work too), then set the stock name as the value of the _HQ_GROUP_ID property.

This will ensure that all messages for a particular stock will always be processed by the same consumer.

28.1. Using Core API

The property name used to identify the message group is `"_HQ_GROUP_ID""` (or the constant `MessageImpl.HDR_GROUP_ID`). Alternatively, you can set `autogroup` to true on the `SessionFactory` which will pick a random unique id.

28.2. Using JMS

The property name used to identify the message group is `JMSXGroupID`.

```
// send 2 messages in the same group to ensure the same
// consumer will receive both
Message message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);

message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);
```

Alternatively, you can set `autogroup` to true on the `HornetQConnectonFactory` which will pick a random unique

id. This can also be set in the `hornetq-jms.xml` file like this:

```
<connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty-connector"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
      <autogroup>true</autogroup>
</connection-factory>
```

Alternatively you can set the group id via the connection factory. All messages sent with producers created via this connection factory will set the `JMSXGroupID` to the specified value on all messages sent. To configure the group id set it on the connection factory in the `hornetq-jms.xml` config file as follows

```
         <connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty-connector"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
      <group-id>Group-0</group-id>
   </connection-factory>
```

28.3. Example

See Section 11.1.29 for an example which shows how message groups are configured and used with JMS.

28.4. Example

See Section 11.1.30 for an example which shows how message groups are configured via a connection factory.

28.5. Clustered Grouping

Using message groups in a cluster is a bit more complex. This is because messages with a particular group id can arrive on any node so each node needs to know about which group id's are bound to which consumer on which node. The consumer handling messages for a particular group id may be on a different node of the cluster, so each node needs to know this information so it can route the message correctly to the node which has that consumer.

To solve this there is the notion of a grouping handler. Each node will have its own grouping handler and when a messages is sent with a group id assigned, the handlers will decide between them which route the message should take.

There are 2 types of handlers; Local and Remote. Each cluster should choose 1 node to have a local grouping handler and all the other nodes should have remote handlers- it's the local handler that actually makes the decsion as to what route should be used, all the other remote handlers converse with this. Here is a sample config for both types of handler, this should be configured in the *hornetq-configuration.xml* file.

```
<grouping-handler name="my-grouping-handler">
   <type>LOCAL</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>

<grouping-handler name="my-grouping-handler">
   <type>REMOTE</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>
```

The *address* attribute refers to a cluster connection and the address it uses, refer to the clustering section on how to configure clusters. The *timeout* attribute referes to how long to wait for a decision to be made, an exception will be thrown during the send if this timeout is reached, this ensures that strict ordering is kept.

The decision as to where a message should be routed to is initially proposed by the node that receives the message. The node will pick a suitable route as per the normal clustered routing conditions, i.e. round robin available queues, use a local queue first and choose a queue that has a consumer. If the proposal is excepted by the grouping handlers the node will route messages to this queue from that point on, if rejected an alternative route will be offered and the node will again route to that queue indefinitely. All other nodes will also route to the queue chosen at proposal time. Once the message arrives at the queue then normal single server message group semantics take over and the message is pinned to a consumer on that queue.

You may have noticed that there is a single point of failure with the single local handler. If this node crashes then no decisions will be able to be made. Any messages sent will be not be delivered and an exception thrown. To avoid this happening Local Handlers can be replicated on another backup node. Simple create your back up node and configure it with the same Local handler.

28.5.1. Clustered Grouping Best Practices

Some best practices should be followed when using clustered grouping:

1. Make sure your consumers are distributed evenly across the different nodes if possible. This is only an issue if you are creating and closing consumers regularly. Since messages are always routed to the same queue once pinned, removing a consumer from this queue may leave it with no consumers meaning the queue will just keep receiving the messages. Avoid closing consumers or make sure that you always have plenty of consumers, i.e., if you have 3 nodes have 3 consumers.

2. Use durable queues if possible. If queues are removed once a group is bound to it, then it is possible that other nodes may still try to route messages to it. This can be avoided by making sure that the queue is deleted by the session that is sending the messages. This means that when the next message is sent it is sent to the node where the queue was deleted meaning a new proposal can succesfully take place. Alternatively you could just start using a different group id.

3. Always make sure that the node that has the Local Grouping Handler is replicated. These means that on failover grouping will still occur.

28.5.2. Clustered Grouping Example

See Section 11.1.6 for an example of how to configure message groups with a HornetQ cluster

29

Pre-Acknowledge Mode

JMS specifies 3 acknowledgement modes:

- `AUTO_ACKNOWLEDGE`
- `CLIENT_ACKNOWLEDGE`
- `DUPS_OK_ACKNOWLEDGE`

However there is another case which is not supported by JMS: In some cases you can afford to lose messages in event of failure, so it would make sense to acknowledge the message on the server *before* delivering it to the client.

This extra mode is supported by HornetQ and will call it *pre-acknowledge* mode.

The disadvantage of acknowledging on the server before delivery is that the message will be lost if the system crashes *after* acknowledging the message on the server but *before* it is delivered to the client. In that case, the message is lost and will not be recovered when the system restart.

Depending on your messaging case, `pre-acknowledgement` mode can avoid extra network traffic and CPU at the cost of coping with message loss.

An example of a use case for pre-acknowledgement is for stock price update messages. With these messages it might be reasonable to lose a message in event of crash, since the next price update message will arrive soon, overriding the previous price.

Note

Please note, that if you use pre-acknowledge mode, then you will lose transactional semantics for messages being consumed, since clearly they are being acknowledged first on the server, not when you commit the transaction. This may be stating the obvious but we like to be clear on these things to avoid confusion!

29.1. Using PRE_ACKNOWLEDGE

This can be configured in the `hornetq-jms.xml` file on the `connection factory` like this:

```
<connection-factory name="ConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty-connector"/>
      </connectors>
      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
      <pre-acknowledge>true</pre-acknowledge>
</connection-factory>
```

Alternatively, to use pre-acknowledgement mode using the JMS API, create a JMS Session with the `HornetQSession.PRE_ACKNOWLEDGE` constant.

```
// messages will be acknowledge on the server *before* being delivered to the client
Session session = connection.createSession(false, HornetQSession.PRE_ACKNOWLEDGE);
```

Or you can set pre-acknowledge directly on the `HornetQConnectionFactory` instance using the setter method.

To use pre-acknowledgement mode using the core API you can set it directly on the `ClientSessionFactory` instance using the setter method.

29.2. Example

See Section 11.1.35 for an example which shows how to use pre-acknowledgement mode with with JMS.

30

Management

HornetQ has an extensive management API that allows a user to modify a server configuration, create new resources (e.g. JMS queues and topics), inspect these resources (e.g. how many messages are currently held in a queue) and interact with it (e.g. to remove messages from a queue). All the operations allows a client to *manage* HornetQ. It also allows clients to subscribe to management notifications.

There are 3 ways to manage HornetQ:

- Using JMX -- JMX is the standard way to manage Java applications
- Using the core API -- management operations are sent to HornetQ server using *core messages*
- Using the JMS API -- management operations are sent to HornetQ server using *JMS messages*

Although there are 3 different ways to manage HornetQ each API supports the same functionality. If it is possible to manage a resource using JMX it is also possible to achieve the same result using Core messages or JMS messages.

This choice depends on your requirements, your application settings and your environment to decide which way suits you best.

30.1. The Management API

Regardless of the way you *invoke* management operations, the management API is the same.

For each *managed resource*, there exists a Java interface describing what can be invoked for this type of resource.

HornetQ exposes its managed resources in 2 packages:

- *Core* resources are located in the `org.hornetq.api.core.management` package
- *JMS* resources are located in the `org.hornetq.api.jms.management` package

The way to invoke a *management operations* depends whether JMX, core messages, or JMS messages are used.

Note

A few management operations requires a `filter` parameter to chose which messages are involved by the operation. Passing `null` or an empty string means that the management operation will be performed on *all messages*.

30.1.1. Core Management API

HornetQ defines a core management API to manage core resources. For full details of the API please consult the javadoc. In summary:

30.1.1.1. Core Server Management

- Listing, creating, deploying and destroying queues

  A list of deployed core queues can be retrieved using the `getQueueNames()` method.

  Core queues can be created or destroyed using the management operations `createQueue()` or `deployQueue()` or `destroyQueue()`)on the `HornetQServerControl` (with the ObjectName `org.hornetq:module=Core,type=Server` or the resource name `core.server`)

  `createQueue` will fail if the queue already exists while `deployQueue` will do nothing.

- Pausing and resuming Queues

  The `QueueControl` can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it's resumed, it'll begin delivering the queued messages, if any.

- Listing and closing remote connections

  Client's remote addresses can be retrieved using `listRemoteAddresses()`. It is also possible to close the connections associated with a remote address using the `closeConnectionsForAddress()` method.

  Alternatively, connection IDs can be listed using `listConnectionIDs()` and all the sessions for a given connection ID can be listed using `listSessions()`.

- Transaction heuristic operations

  In case of a server crash, when the server restarts, it it possible that some transaction requires manual intervention. The `listPreparedTransactions()` method lists the transactions which are in the prepared states (the transactions are represented as opaque Base64 Strings.) To commit or rollback a given prepared transaction, the `commitPreparedTransaction`() or `rollbackPreparedTransaction()` method can be used to resolve heuristic transactions. Heuristically completed transactions can be listed using the `listHeuristicCommittedTransactions()` and `listHeuristicRolledBackTransactions` methods.

- Enabling and resetting Message counters

  Message counters can be enabled or disabled using the `enableMessageCounters()` or `disableMessageCounters()` method. To reset message counters, it is possible to invoke `resetAllMessageCounters()` and `resetAllMessageCounterHistories()` methods.

- Retrieving the server configuration and attributes

  The `HornetQServerControl` exposes HornetQ server configuration through all its attributes (e.g. `getVersion()` method to retrieve the server's version, etc.)

30.1.1.2. Core Address Management

Core addresses can be managed using the `AddressControl` class (with the ObjectName `org.hornetq:module=Core,type=Address,name="<the address name>"` or the resource name `core.address.<the address name>`).

- Modifying roles and permissions for an address

  You can add or remove roles associated to a queue using the `addRole()` or. `removeRole()` methods. You can list all the roles associated to the queue with the `getRoles()` method

30.1.1.3. Core Queue Management

The bulk of the core management API deals with core queues. The `QueueControl` class defines the Core queue management operations (with the ObjectName `org.hornetq:module=Core,type=Queue,address="<the bound address>",name="<the queue name>"` or the resource name `core.queue.<the queue name>`).

Most of the management operations on queues take either a single message ID (e.g. to remove a single message) or a filter (e.g. to expire all messages with a given property.)

- Expiring, sending to a dead letter address and moving messages

  Messages can be expired from a queue by using the `expireMessages()` method. If an expiry address is defined, messages will be be sent to it, otherwise they are discarded. The queue's expiry address can be set with the `setExpiryAddress()` method.

  Messages can also be sent to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded. The queue's dead letter address can be set with the `setDeadLetterAddress()` method.

  Messages can also be moved from a queue to another queue by using the `moveMessages()` method.

- Listing and removing messages

  Messages can be listed from a queue by using the `listMessages()` method which returns an array of `Map`, one `Map` for each message.

  Messages can also be removed from the queue by using the `removeMessages()` method which returns a `boolean` for the single message ID variant or the number of removed messages for the filter variant. The `removeMessages()` method takes a `filter` argument to remove only filtered messages. Setting the filter to an empty string will in effect remove all messages.

- Counting messages

  The number of messages in a queue is returned by the `getMessageCount()` method. Alternatively, the `countMessages()` will return the number of messages in the queue which *match a given filter*

- Changing message priority

  The message priority can be changed by using the `changeMessagesPriority()` method which returns a `boolean` for the single message ID variant or the number of updated messages for the filter variant.

- Message counters

  Message counters can be listed for a queue with the `listMessageCounter()` and `listMessageCounterHistory()` methods (see Section 30.6). The message counters can also be reset for a single queue using the `resetMessageCounter()` method.

- Retrieving the queue attributes

  The `QueueControl` exposes Core queue settings through its attributes (e.g. `getFilter()` to retrieve the queue's filter if it was created with one, `isDurable()` to know whether the queue is durable or not, etc.)

- Pausing and resuming Queues

  The `QueueControl` can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it's resume, it'll begin delivering the queued messages, if any.

30.1.1.4. Other Core Resources Management

HornetQ allows to start and stop its remote resources (acceptors, diverts, bridges, etc.) so that a server can be taken off line for a given period of time without stopping it completely (e.g. if other management operations must be performed such as resolving heuristic transactions). These resources are:

- Acceptors

  They can be started or stopped using the `start()` or. `stop()` method on the `AcceptorControl` class (with the ObjectName `org.hornetq:module=Core,type=Acceptor,name="<the acceptor name>"` or the resource name `core.acceptor.<the address name>`). The acceptors parameters can be retrieved using the `AcceptorControl` attributes (see Section 16.1)

- Diverts

  They can be started or stopped using the `start()` or `stop()` method on the `DivertControl` class (with the ObjectName `org.hornetq:module=Core,type=Divert,name=<the divert name>` or the resource name `core.divert.<the divert name>`). Diverts parameters can be retrieved using the `DivertControl` attributes (see Chapter 35)

- Bridges

  They can be started or stopped using the `start()` (resp. `stop()`) method on the `BridgeControl` class (with the ObjectName `org.hornetq:module=Core,type=Bridge,name="<the bridge name>"` or the resource name `core.bridge.<the bridge name>`). Bridges parameters can be retrieved using the `BridgeControl` attributes (see Chapter 36)

- Broadcast groups

  They can be started or stopped using the `start()` or `stop()` method on the `BroadcastGroupControl` class (with the ObjectName `org.hornetq:module=Core,type=BroadcastGroup,name="<the broadcast group name>"` or the resource name `core.broadcastgroup.<the broadcast group name>`). Broadcast groups parameters can be retrieved using the `BroadcastGroupControl` attributes (see Section 38.2.1)

- Discovery groups

They can be started or stopped using the `start()` or `stop()` method on the `DiscoveryGroupControl` class (with the ObjectName `org.hornetq:module=Core,type=DiscoveryGroup,name="<the discovery group name>"` or the resource name `core.discovery.<the discovery group name>`). Discovery groups parameters can be retrieved using the `DiscoveryGroupControl` attributes (see Section 38.2.2)

- Cluster connections

  They can be started or stopped using the `start()` or `stop()` method on the `ClusterConnectionControl` class (with the ObjectName `org.hornetq:module=Core,type=ClusterConnection,name="<the cluster connection name>"` or the resource name `core.clusterconnection.<the cluster connection name>`). Cluster connections parameters can be retrieved using the `ClusterConnectionControl` attributes (see Section 38.3.1)

30.1.2. JMS Management API

HornetQ defines a JMS Management API to manage JMS *administrated objects* (i.e. JMS queues, topics and connection factories).

30.1.2.1. JMS Server Management

JMS Resources (connection factories and destinations) can be created using the `JMSServerControl` class (with the ObjectName `org.hornetq:module=JMS,type=Server` or the resource name `jms.server`).

- Listing, creating, destroying connection factories

  Names of the deployed connection factories can be retrieved by the `getConnectionFactoryNames()` method.

  JMS connection factories can be created or destroyed using the `createConnectionFactory()` methods or `destroyConnectionFactory()` methods. These connection factories are bound to JNDI so that JMS clients can look them up. If a graphical console is used to create the connection factories, the transport parameters are specified in the text fied input as a comma-separated list of key=value (e.g. `key1=10, key2="value", key3=false`). If there are multiple transports defined, you need to enclose the key/value pairs between curly braces. For example `{key=10}, {key=20}`. In that case, the first `key` will be associated to the first transport configuration and the second `key` will be associated to the second transport configuration (see Chapter 16 for a list of the transport parameters)

- Listing, creating, destroying queues

  Names of the deployed JMS queues can be retrieved by the `getQueueNames()` method.

  JMS queues can be created or destroyed using the `createQueue()` methods or `destroyQueue()` methods. These queues are bound to JNDI so that JMS clients can look them up

- Listing, creating/destroying topics

  Names of the deployed topics can be retrieved by the `getTopicNames()` method.

  JMS topics can be created or destroyed using the `createTopic()` or `destroyTopic()` methods. These topics are bound to JNDI so that JMS clients can look them up

- Listing and closing remote connections

  JMS Clients remote addresses can be retrieved using `listRemoteAddresses()`. It is also possible to close the connections associated with a remote address using the `closeConnectionsForAddress()` method.

  Alternatively, connection IDs can be listed using `listConnectionIDs()` and all the sessions for a given connection ID can be listed using `listSessions()`.

30.1.2.2. JMS ConnectionFactory Management

JMS Connection Factories can be managed using the `ConnectionFactoryControl` class (with the ObjectName `org.hornetq:module=JMS,type=ConnectionFactory,name="<the connection factory name>"` or the resource name `jms.connectionfactory.<the connection factory name>`).

- Retrieving connection factory attributes

  The `ConnectionFactoryControl` exposes JMS ConnectionFactory configuration through its attributes (e.g. `getConsumerWindowSize()` to retrieve the consumer window size for flow control, `isBlockOnNonDurableSend()` to know wether the producers created from the connection factory will block or not when sending non-durable messages, etc.)

30.1.2.3. JMS Queue Management

JMS queues can be managed using the `JMSQueueControl` class (with the ObjectName `org.hornetq:module=JMS,type=Queue,name="<the queue name>"` or the resource name `jms.queue.<the queue name>`).

The management operations on a JMS queue are very similar to the operations on a core queue.

- Expiring, sending to a dead letter address and moving messages

  Messages can be expired from a queue by using the `expireMessages()` method. If an expiry address is defined, messages will be be sent to it, otherwise they are discarded. The queue's expiry address can be set with the `setExpiryAddress()` method.

  Messages can also be sent to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded. The queue's dead letter address can be set with the `setDeadLetterAddress()` method.

  Messages can also be moved from a queue to another queue by using the `moveMessages()` method.

- Listing and removing messages

  Messages can be listed from a queue by using the `listMessages()` method which returns an array of `Map`, one `Map` for each message.

  Messages can also be removed from the queue by using the `removeMessages()` method which returns a `boolean` for the single message ID variant or the number of removed messages for the filter variant. The `removeMessages()` method takes a `filter` argument to remove only filtered messages. Setting the filter to an

empty string will in effect remove all messages.

- Counting messages

  The number of messages in a queue is returned by the `getMessageCount()` method. Alternatively, the `countMessages()` will return the number of messages in the queue which *match a given filter*

- Changing message priority

  The message priority can be changed by using the `changeMessagesPriority()` method which returns a `boolean` for the single message ID variant or the number of updated messages for the filter variant.

- Message counters

  Message counters can be listed for a queue with the `listMessageCounter()` and `listMessageCounterHistory()` methods (see Section 30.6)

- Retrieving the queue attributes

  The `JMSQueueControl` exposes JMS queue settings through its attributes (e.g. `isTemporary()` to know wether the queue is temporary or not, `isDurable()` to know wether the queue is durable or not, etc.)

- Pausing and resuming queues

  The `JMSQueueControl` can pause and resume the underlying queue. When the queue is paused it will continue to receive messages but will not deliver them. When resumed again it will deliver the enqueued messages, if any.

30.1.2.4. JMS Topic Management

JMS Topics can be managed using the `TopicControl` class (with the ObjectName `org.hornetq:module=JMS,type=Topic,name="<the topic name>"` or the resource name `jms.topic.<the topic name>`).

- Listing subscriptions and messages

  JMS topics subscriptions can be listed using the `listAllSubscriptions()`, `listDurableSubscriptions()`, `listNonDurableSubscriptions()` methods. These methods return arrays of `Object` representing the subscriptions information (subscription name, client ID, durability, message count, etc.). It is also possible to list the JMS messages for a given subscription with the `listMessagesForSubscription()` method.

- Dropping subscriptions

  Durable subscriptions can be dropped from the topic using the `dropDurableSubscription()` method.

- Counting subscriptions messages

  The `countMessagesForSubscription()` method can be used to know the number of messages held for a given subscription (with an optional message selector to know the number of messages matching the selector)

30.2. Using Management Via JMX

HornetQ can be managed using JMX [http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/].

The management API is exposed by HornetQ using MBeans interfaces. HornetQ registers its resources with the domain `org.hornetq`.

For example, the `ObjectName` to manage a JMS Queue `exampleQueue` is:

```
org.hornetq:module=JMS,type=Queue,name="exampleQueue"
```

and the MBean is:

```
org.hornetq.api.jms.management.JMSQueueControl
```

The MBean's `ObjectName` are built using the helper class `org.hornetq.api.core.management.ObjectNameBuilder`. You can also use `jconsole` to find the `ObjectName` of the MBeans you want to manage.

Managing HornetQ using JMX is identical to management of any Java Applications using JMX. It can be done by reflection or by creating proxies of the MBeans.

30.2.1. Configuring JMX

By default, JMX is enabled to manage HornetQ. It can be disabled by setting `jmx-management-enabled` to `false` in `hornetq-configuration.xml`:

```
<!-- false to disable JMX management for HornetQ -->
<jmx-management-enabled>false</jmx-management-enabled>
```

If JMX is enabled, HornetQ can be managed locally using `jconsole`. Remote connections to JMX are not enabled by default for security reasons. Please refer to Java Management guide [http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote] to configure the server for remote management (system properties must be set in `run.sh` or `run.bat` scripts).

By default, HornetQ server uses the JMX domain "org.hornetq". To manage several HornetQ servers from the *same* MBeanServer, the JMX domain can be configured for each individual HornetQ server by setting `jmx-domain` in `hornetq-configuration.xml`:

```
<!-- use a specific JMX domain for HornetQ MBeans -->
<jmx-domain>my.org.hornetq</jmx-domain>
```

30.2.1.1. MBeanServer configuration

When HornetQ is run in standalone, it uses the Java Virtual Machine's `Platform MBeanServer` to register its MBeans. This is configured in JBoss Microcontainer Beans file (see Section 6.7):

```
<!-- MBeanServer -->
<bean name="MBeanServer" class="javax.management.MBeanServer">
   <constructor factoryClass="java.lang.management.ManagementFactory"
                         factoryMethod="getPlatformMBeanServer" />
</bean>
```

When it is integrated in JBoss AS 5+, it uses the Application Server's own MBean Server so that it can be managed using AS 5's jmx-console:

```
<!-- MBeanServer -->
<bean name="MBeanServer" class="javax.management.MBeanServer">
   <constructor factoryClass="org.jboss.mx.util.MBeanServerLocator"
                         factoryMethod="locateJBoss" />
</bean>
```

30.2.2. Example

See Section 11.1.21 for an example which shows how to use a remote connection to JMX and MBean proxies to manage HornetQ.

30.3. Using Management Via Core API

The core management API in HornetQ is called by sending Core messages to a special address, the *management address*.

Management messages are regular Core messages with well-known properties that the server needs to understand to interact with the management API:

- The name of the managed resource
- The name of the management operation
- The parameters of the management operation

When such a management message is sent to the management address, HornetQ server will handle it, extract the information, invoke the operation on the managed resources and send a *management reply* to the management message's reply-to address (specified by `ClientMessageImpl.REPLYTO_HEADER_NAME`).

A `ClientConsumer` can be used to consume the management reply and retrieve the result of the operation (if any) stored in the reply's body. For portability, results are returned as a JSON [http://json.org] String rather than Java Serialization (the `org.hornetq.api.core.management.ManagementHelper` can be used to convert the JSON string to Java objects).

These steps can be simplified to make it easier to invoke management operations using Core messages:

1. Create a `ClientRequestor` to send messages to the management address and receive replies
2. Create a `ClientMessage`
3. Use the helper class `org.hornetq.api.core.management.ManagementHelper` to fill the message with the management properties
4. Send the message using the `ClientRequestor`
5. Use the helper class `org.hornetq.api.core.management.ManagementHelper` to retrieve the operation result from the management reply

For example, to find out the number of messages in the core queue `exampleQueue`:

```
ClientSession session = ...
ClientRequestor requestor = new ClientRequestor(session, "jms.queue.hornetq.management");
ClientMessage message = session.createMessage(false);
ManagementHelper.putAttribute(message, "core.queue.exampleQueue", "messageCount");
ClientMessage reply = requestor.request(m);
int count = (Integer) ManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

Management operation name and parameters must conform to the Java interfaces defined in the `management` packages.

Names of the resources are built using the helper class `org.hornetq.api.core.management.ResourceNames` and are straightforward (`core.queue.exampleQueue` for the Core Queue `exampleQueue`, `jms.topic.exampleTopic` for the JMS Topic `exampleTopic`, etc.).

30.3.1. Configuring Core Management

The management address to send management messages is configured in `hornetq-configuration.xml`:

```
<management-address>jms.queue.hornetq.management</management-address>
```

By default, the address is `jms.queue.hornetq.management` (it is prepended by "jms.queue" so that JMS clients can also send management messages).

The management address requires a *special* user permission `manage` to be able to receive and handle management messages. This is also configured in hornetq-configuration.xml:

```
<!-- users with the admin role will be allowed to manage -->
<!-- HornetQ using management messages        -->
<security-setting match="jms.queue.hornetq.management">
   <permission type="manage" roles="admin" />
</security-setting>
```

30.4. Using Management Via JMS

Using JMS messages to manage HornetQ is very similar to using core API.

An important difference is that JMS requires a JMS queue to send the messages to (instead of an address for the core API).

The *management queue* is a special queue and needs to be instantiated directly by the client:

```
Queue managementQueue = HornetQJMSClient.createQueue("hornetq.management");
```

All the other steps are the same than for the Core API but they use JMS API instead:

1. create a `QueueRequestor` to send messages to the management address and receive replies
2. create a `Message`
3. use the helper class `org.hornetq.api.jms.management.JMSManagementHelper` to fill the message with the management properties
4. send the message using the `QueueRequestor`
5. use the helper class `org.hornetq.api.jms.management.JMSManagementHelper` to retrieve the operation result from the management reply

For example, to know the number of messages in the JMS queue `exampleQueue`:

```
Queue managementQueue = HornetQJMSClient.createQueue("hornetq.management");

QueueSession session = ...
QueueRequestor requestor = new QueueRequestor(session, managementQueue);
connection.start();
Message message = session.createMessage();
JMSManagementHelper.putAttribute(message, "jms.queue.exampleQueue", "messageCount");
Message reply = requestor.request(message);
int count = (Integer)JMSManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

30.4.1. Configuring JMS Management

Whether JMS or the core API is used for management, the configuration steps are the same (see Section 30.3.1).

30.4.2. Example

See Section 11.1.25 for an example which shows how to use JMS messages to manage HornetQ server.

30.5. Management Notifications

HornetQ emits *notifications* to inform listeners of potentially interesting events (creation of new resources, security violation, etc.).

These notifications can be received by 3 different ways:

- JMX notifications
- Core messages
- JMS messages

30.5.1. JMX Notifications

If JMX is enabled (see Section 30.2.1), JMX notifications can be received by subscribing to 2 MBeans:

- `org.hornetq:module=Core,type=Server` for notifications on *Core* resources
- `org.hornetq:module=JMS,type=Server` for notifications on *JMS* resources

30.5.2. Core Messages Notifications

HornetQ defines a special *management notification address*. Core queues can be bound to this address so that clients will receive management notifications as Core messages

A Core client which wants to receive management notifications must create a core queue bound to the management notification address. It can then receive the notifications from its queue.

Notifications messages are regular core messages with additional properties corresponding to the notification (its type, when it occurred, the resources which were concerned, etc.).

Since notifications are regular core messages, it is possible to use message selectors to filter out notifications and receives only a subset of all the notifications emitted by the server.

30.5.2.1. Configuring The Core Management Notification Address

the management notification address to receive management notifications is configured in `hornetq-configuration.xml`:

```
<management-notification-address>hornetq.notifications</management-notification-address>
```

By default, the address is `hornetq.notifications`.

30.5.3. JMS Messages Notifications

HornetQ's notifications can also be received using JMS messages.

It is similar to receiving notifications using Core API but an important difference is that JMS requires a JMS Destination to receive the messages (preferably a Topic):

```
Topic notificationsTopic = HornetQJMSClient.createHornetQTopic("hornetq.notifications", "hornetq.notif
```

Once the notification topic is created, you can receive messages from it or set a `MessageListener`:

```
Topic notificationsTopic = HornetQJMSClient.createHornetQTopic("hornetq.notifications", "hornetq.notif

Session session = ...
MessageConsumer notificationConsumer = session.createConsumer(notificationsTopic);
   notificationConsumer.setMessageListener(new MessageListener()
   {
      public void onMessage(Message notif)
      {
         System.out.println("------------------------");
         System.out.println("Received notification:");
         try
         {
            Enumeration propertyNames = notif.getPropertyNames();
            while (propertyNames.hasMoreElements())
            {
               String propertyName = (String)propertyNames.nextElement();
               System.out.format("  %s: %s\n", propertyName, notif.getObjectProperty(propertyName));
            }
         }
         catch (JMSException e)
         {
         }
         System.out.println("------------------------");
      }
   });
```

30.5.4. Example

See Section 11.1.26 for an example which shows how to use a JMS `MessageListener` to receive management notifications from HornetQ server.

30.6. Message Counters

Message counters can be used to obtain information on queues *over time* as HornetQ keeps a history on queue metrics.

They can be used to show *trends* on queues. For example, using the management API, it would be possible to query the number of messages in a queue at regular interval. However, this would not be enough to know if the queue is used: the number of messages can remain constant because nobody is sending or receiving messages from the queue or because there are as many messages sent to the queue than messages consumed from it. The number of messages in the queue remains the same in both cases but its use is widely different.

Message counters gives additional information about the queues:

- `count`

  The *total* number of messages added to the queue since the server was started

- `countDelta`

  the number of messages added to the queue *since the last message counter update*

- `depth`

  The *current* number of messages in the queue

- `depthDelta`

  The *overall* number of messages added/removed from the queue *since the last message counter update*. For example, if `depthDelta` is equal to `-10` this means that overall 10 messages have been removed from the queue (e.g. 2 messages were added and 12 were removed)

- `lastAddTimestamp`

  The timestamp of the last time a message was added to the queue

- `udpateTimestamp`

  The timestamp of the last message counter update

30.6.1. Configuring Message Counters

By default, message counters are disabled as it might have a small negative effect on memory.

To enable message counters, you can set it to `true` in `hornetq-configuration.xml`:

```
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keeps a history of the queue metrics (10 days by default) and samples all the queues at regular interval (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in `hornetq-configuration.xml`:

```
<!-- keep history for a week -->
<message-counter-max-day-history>7</message-counter-max-day-history>
<!-- sample the queues every minute (60000ms) -->
<message-counter-sample-period>60000</message-counter-sample-period>
```

Message counters can be retrieved using the Management API. For example, to retrieve message counters on a JMS Queue using JMX:

```

```

```
// retrieve a connection to HornetQ's MBeanServer
MBeanServerConnection mbsc = ...
JMSQueueControlMBean queueControl = (JMSQueueControl)MBeanServerInvocationHandler.newProxyInstance(mbsc,
   on,
   JMSQueueControl.class,
   false);
// message counters are retrieved as a JSON String
String counters = queueControl.listMessageCounter();
// use the MessageCounterInfo helper class to manipulate message counters more easily
MessageCounterInfo messageCounter = MessageCounterInfo.fromJSON(counters);
System.out.format("%s message(s) in the queue (since last sample: %s)\n",
   counter.getDepth(),
   counter.getDepthDelta());
```

30.6.2. Example

See Section 11.1.27 for an example which shows how to use message counters to retrieve information on a JMS `Queue`.

31

Security

This chapter describes how security works with HornetQ and how you can configure it. To disable security completely simply set the `security-enabled` property to false in the `hornetq-configuration.xml` file.

For performance reasons security is cached and invalidated every so long. To change this period set the property `security-invalidation-interval`, which is in milliseconds. The default is `10000` ms.

31.1. Role based security for addresses

HornetQ contains a flexible role-based security model for applying security to queues, based on their addresses.

As explained in Chapter 8, HornetQ core consists mainly of sets of queues bound to addresses. A message is sent to an address and the server looks up the set of queues that are bound to that address, the server then routes the message to those set of queues.

HornetQ allows sets of permissions to be defined against the queues based on their address. An exact match on the address can be used or a wildcard match can be used using the wildcard characters '`#`' and '`*`'.

Seven different permissions can be given to the set of queues which match the address. Those permissions are:

- `createDurableQueue`. This permission allows the user to create a durable queue under matching addresses.
- `deleteDurableQueue`. This permission allows the user to delete a durable queue under matching addresses.
- `createTempQueue`. This permission allows the user to create a temporary queue under matching addresses.
- `deleteTempQueue`. This permission allows the user to delete a temporarry queue under matching addresses.
- `send`. This permission allows the user to send a message to matching addresses.
- `consume`. This permission allows the user to consume a message from a queue bound to matching addresses.
- `manage`. This permission allows the user to invoke management operations by sending management messages to the management address.

For each permission, a list of roles who are granted that permission is specified. If the user has any of those roles, he/she will be granted that permission for that set of addresses.

Let's take a simple example, here's a security block from `hornetq-configuration.xml` or `hornetq-queues.xml` file:

```
<security-setting match="globalqueues.europe.#">
```

```
    <permission type="createDurableQueue" roles="admin"/>
    <permission type="deleteDurableQueue" roles="admin"/>
    <permission type="createTempQueue" roles="admin, guest, europe-users"/>
    <permission type="deleteTempQueue" roles="admin, guest, europe-users"/>
    <permission type="send" roles="admin, europe-users"/>
    <permission type="consume" roles="admin, europe-users"/>
</security-setting>
```

The '#' character signifies "any sequence of words". Words are delimited by the '.' character. For a full description of the wildcard syntax please see Chapter 13. The above security block applies to any address that starts with the string "globalqueues.europe.":

Only users who have the `admin` role can create or delete durable queues bound to an address that starts with the string "globalqueues.europe."

Only users who have the `admin` role can create or delete durable queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles `admin`, `guest`, or `europe-users` can create or delete temporary queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles `admin` or `europe-users` can send messages to these addresses or consume messages from queues bound to an address that starts with the string "globalqueues.europe."

The mapping between a user and what roles they have is handled by the security manager. HornetQ ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss Application Server security.

For more information on configuring the security manager, please see Section 31.4.

There can be zero or more `security-setting` elements in each xml file. Where more than one match applies to a set of addresses the *more specific* match takes precedence.

Let's look at an example of that, here's another `security-setting` block:

```
<security-setting match="globalqueues.europe.orders.#">
    <permission type="send" roles="europe-users"/>
    <permission type="consume" roles="europe-users"/>
</security-setting>
```

In this `security-setting` block the match 'globalqueues.europe.orders.#' is more specific than the previous match 'globalqueues.europe.#'. So any addresses which match 'globalqueues.europe.orders.#' will take their security settings *only* from the latter security-setting block.

Note that settings are not inherited from the former block. All the settings will be taken from the more specific matching block, so for the address 'globalqueues.europe.orders.plastics' the only permissions that exist are `send` and `consume` for the role europe-users. The permissions `createDurableQueue`, `deleteDurableQueue`, `createTempQueue`, `deleteTempQueue` are not inherited from the other security-setting block.

By not inheriting permissions, it allows you to effectively deny permissions in more specific security-setting blocks by simply not specifying them. Otherwise it would not be possible to deny permissions in sub-groups of addresses.

31.2. Secure Sockets Layer (SSL) Transport

When messaging clients are connected to servers, or servers are connected to other servers (e.g. via bridges) over an untrusted network then HornetQ allows that traffic to be encrypted using the Secure Sockets Layer (SSL) transport.

For more information on configuring the SSL transport, please see Chapter 16.

31.3. Basic user credentials

HornetQ ships with a security manager implementation that reads user credentials, i.e. user names, passwords and role information from an xml file on the classpath called `hornetq-users.xml`. This is the default security manager.

If you wish to use this security manager, then users, passwords and roles can easily be added into this file.

Let's take a look at an example file:

```
<configuration xmlns="urn:hornetq"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:hornetq ../schemas/hornetq-users.xsd ">

    <defaultuser name="guest" password="guest">
        <role name="guest"/>
    </defaultuser>

    <user name="tim" password="marmite">
        <role name="admin"/>
    </user>

    <user name="andy" password="doner_kebab">
        <role name="admin"/>
        <role name="guest"/>
    </user>

    <user name="jeff" password="camembert">
        <role name="europe-users"/>
        <role name="guest"/>
    </user>

</configuration>
```

The first thing to note is the element `defaultuser`. This defines what user will be assumed when the client does not specify a username/password when creating a session. In this case they will be the user `guest` and have the role also called `guest`. Multiple roles can be specified for a default user.

We then have three more users, the user `tim` has the role `admin`. The user `andy` has the roles `admin` and `guest`, and the user `jeff` has the roles `europe-users` and `guest`.

31.4. Changing the security manager

If you do not want to use the default security manager then you can specify a different one by editing the file `hor-`

`netq-beans.xml` (or `hornetq-jboss-beans.xml` if you're running JBoss Application Server) and changing the class for the `HornetQSecurityManager` bean.

Let's take a look at a snippet from the default beans file:

```
<bean name="HornetQSecurityManager"
      class="org.hornetq.spi.core.security.HornetQSecurityManagerImpl">
    <start ignored="true"/>
    <stop ignored="true"/>
</bean>
```

The class `org.hornetq.spi.core.security.HornetQSecurityManagerImpl` is the default security manager that is used by the standalone server.

HornetQ ships with two other security manager implementations you can use off-the-shelf; one a JAAS security manager and another for integrating with JBoss Application Sever security, alternatively you could write your own implementation by implementing the `org.hornetq.core.security.SecurityManager` interface, and specifying the classname of your implementation in the file `hornetq-beans.xml` (or `hornetq-jboss-beans.xml` if you're running JBoss Application Server).

These two implementations are discussed in the next two sections.

31.5. JAAS Security Manager

JAAS stands for 'Java Authentication and Authorization Service' and is a standard part of the Java platform. It provides a common API for security authentication and authorization, allowing you to plugin your pre-built implementations.

To configure the JAAS security manager to work with your pre-built JAAS infrastructure you need to specify the security manager as a `JAASSecurityManager` in the beans file. Here's an example:

```
&lt;bean name="HornetQSecurityManager"
      class="org.hornetq.integration.jboss.security.JAASSecurityManager"&gt;
    &lt;start ignored="true"/&gt;
    &lt;stop ignored="true"/&gt;

    &lt;property name="ConfigurationName"&gt;org.hornetq.jms.example.ExampleLoginModule&lt;/property&gt;
    &lt;property name="Configuration"&gt;
       &lt;inject bean="ExampleConfiguration"/&gt;
    &lt;/property&gt;
    &lt;property name="CallbackHandler"&gt;
       &lt;inject bean="ExampleCallbackHandler"/&gt;
    &lt;/property&gt;
&lt;/bean&gt;
```

Note that you need to feed the JAAS security manager with three properties:

- ConfigurationName: the name of the `LoginModule` implementation that JAAS must use
- Configuration: the `Configuration` implementation used by JAAS

- CallbackHandler: the `CallbackHandler` implementation to use if user interaction are required

31.5.1. Example

See Section 11.1.19 for an example which shows how HornetQ can be configured to use JAAS.

31.6. JBoss AS Security Manager

The JBoss AS security manager is used when running HornetQ inside the JBoss Application server. This allows tight integration with the JBoss Application Server's security model.

The class name of this security manager is `org.hornetq.integration.jboss.security.JBossASSecurityManager`

Take a look at one of the default `hornetq-jboss-beans.xml` files for JBoss Application Server that are bundled in the distribution for an example of how this is configured.

31.7. Changing the username/password for clustering

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The username/password they use for this should always be changed from the installation default to prevent a security risk.

Please see Chapter 30 for instructions on how to do this.

32

Application Server Integration and Java EE

HornetQ can be easily installed in JBoss Application Server 4 or later. For details on installing HornetQ in the JBoss Application Server please refer to quick-start guide.

Since HornetQ also provides a JCA adaptor, it should also be possible to integrate HornetQ as a JMS provider in other JEE compliant app servers. For instructions on how to integrate a remote JCA adaptor into another application sever, please consult the other application server's instructions.

A JCA Adapter basically controls the inflow of messages to Message Driven Beans and the outflow of messages sent from other JEE components, e.g. EJBs and Servlets.

This section explains the basics behind configuring the different JEE components in the AS.

32.1. Configuring Message Driven Beans

The delivery of messages to an MDB using HornetQ is configured on the JCA Adapter via a configuration file `ra.xml` which can be found under in the `jms-ra.rar` archive of directory. By default this is configured to consume messages using an InVM connector from the instance of HornetQ running within the application server. A full list of what is configurable is found later in this chapter.

All MDB's however need to have the destination type and the destination configured. The following example shows how this can be done via annotations.

```
@MessageDriven(name = "MDBExample",
               activationConfig =
                     {
                        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "java
                        @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/te
                     })
@ResourceAdapter("hornetq-ra.rar")
public class MDBExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

In this example you can see that the MDB will consume messages from a queue that is mapped into JNDI with the binding `queue/testQueue`. This queue must be preconfigured in the usual way using the HornetQ configuration files.

The `ResourceAdapter` annotation is used to specify which adaptor should be used. To use this you will need to import `org.jboss.ejb3.annotation.ResourceAdapter` which can be found in the `jboss-ejb3-ext-api.jar` which can be found in the jboss repository. Alternatively you can add use a deployment descriptor and add something like the following to `jboss.xml`

```
<message-driven>
   <ejb-name>ExampleMDB</ejb-name>
   <resource-adapter-name>quartz-ra.rar</resource-adapter-name>
</message-driven>
```

You can also rename the hornetq-ra.rar directory to jms-ra.rar and neither the annotation or the extra descriptor information will be needed. If you do this you will need to edit the `jms-ds.xml` datasource file and change `rar-name` element.

All the examples shipped with the HornetQ distribution use the annotation.

32.1.1. Using Container Managed Transactions

When an MDB is using Container Managed Transactions (CMT), the delivery of the message is done within the scope of a JTA transaction. The commit or rollback of this transaction is controlled by the container itself. If the transaction is rolled back then the message delivery semantics will kick in (by default this is to try and redeliver the message up to 10 times before sending to a DLQ). Using annotations this would be configured as follows:

```
@MessageDriven(name = "MDB_CMP_TxRequiredExample",
               activationConfig =
                     {
                        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "java
                        @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/te
                     })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_CMP_TxRequiredExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

The `TransactionManagement` annotation tells the container to treat this MDB to use Container Managed Persistence. The `TransactionAttribute` annotation tells the container that a JTA transaction is required for this MDB. Note that the only other valid value for this is `TransactionAttributeType.NOT_SUPPORTED` which tells the container that this MDB does not support JTA transactions and one should not be created.

It is also possible to inform the container that it must rollback the transaction by calling `setRollbackOnly` on the `MessageDrivenContext`. The code for this would look something like:

```
   @Resource
   MessageDrivenContext ctx;

   public void onMessage(Message message)
   {
      try
      {
         //something here fails
      }
      catch (Exception e)
      {
         ctx.setRollbackOnly();
      }
   }
```

If you don't want the over head of an xa transaction being created every time but you would still like the message delivered within a transaction (i.e. you are only using a JMS resource) then you can configure the MDB to use a

local transaction. This would be configured as such:

```
@MessageDriven(name = "MDB_CMP_TxLocalExample",
               activationConfig =
                     {
                           @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "j
                           @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue
                           @ActivationConfigProperty(propertyName = "useLocalTx", propertyValue = "true")
                     })
@TransactionManagement(value = TransactionManagementType.CONTAINER)
@TransactionAttribute(value = TransactionAttributeType.NOT_SUPPORTED)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_CMP_TxLocalExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

32.1.2. Using Bean Managed Transactions

Message driven beans can also be configured to use Bean Managed Transactions (BMT). In this case a User Transaction is created. This would be configured as follows:

```
@MessageDriven(name = "MDB_BMPExample",
               activationConfig =
                     {
                        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "java
                        @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/te
                        @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Dups
                     })
@TransactionManagement(value= TransactionManagementType.BEAN)
@ResourceAdapter("hornetq-ra.rar")
public class MDB_BMPExample implements MessageListener
{
   public void onMessage(Message message)
}
```

When using Bean Managed Transactions the message delivery to the MDB will occur outside the scope of the user transaction and use the acknowledge mode specified by the user with the `acknowledgeMode` property. There are only 2 acceptable values for this `Auto-acknowledge` and `Dups-ok-acknowledge`. Please note that because the message delivery is outside the scope of the transaction a failure within the MDB will not cause the message to be redelivered.

A user would control the lifecycle of the transaction something like the following:

```
   @Resource
   MessageDrivenContext ctx;

   public void onMessage(Message message)
   {
      UserTransaction tx;
      try
      {
         TextMessage textMessage = (TextMessage)message;

         String text = textMessage.getText();

         UserTransaction tx = ctx.getUserTransaction();

         tx.begin();
```

```
         //do some stuff within the transaction
         tx.commit();

      }
      catch (Exception e)
      {
         tx.rollback();
      }
   }
```

32.1.3. Using Message Selectors with MDB's

It is also possible to use MDB's with message selectors. To do this simple define your message selector as follows:

```
@MessageDriven(name = "MDBMessageSelectorExample",
               activationConfig =
                     {
                        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "java
                        @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/te
                        @ActivationConfigProperty(propertyName = "messageSelector", propertyValue = "colo
                     })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDBMessageSelectorExample implements MessageListener
{
   public void onMessage(Message message)....
}
```

32.2. Sending Messages from within JEE components

The JCA adapter can also be used for sending messages. The Connection Factory to use is configured by default in the `jms-ds.xml` file and is mapped to `java:/JmsXA`. Using this from within a JEE component will mean that the sending of the message will be done as part of the JTA transaction being used by the component.

This means that if the sending of the message fails the overall transaction would rollback and the message redelivered. Heres an example of this from within an MDB:

```
@MessageDriven(name = "MDBMessageSendTxExample",
               activationConfig =
                     {
                        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "java
                        @ActivationConfigProperty(propertyName = "destination", propertyValue = "queue/te
                     })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
@ResourceAdapter("hornetq-ra.rar")
public class MDBMessageSendTxExample implements MessageListener
{
   @Resource(mappedName = "java:/JmsXA")
   ConnectionFactory connectionFactory;

   @Resource(mappedName = "queue/replyQueue")
   Queue replyQueue;

   public void onMessage(Message message)
   {
```

```
      Connection conn = null;
      try
      {
         //Step 9. We know the client is sending a text message so we cast
         TextMessage textMessage = (TextMessage)message;

         //Step 10. get the text from the message.
         String text = textMessage.getText();

         System.out.println("message " + text);

         conn = connectionFactory.createConnection();

         Session sess = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

         MessageProducer producer = sess.createProducer(replyQueue);

         producer.send(sess.createTextMessage("this is a reply"));

      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         if(conn != null)
         {
            try
            {
               conn.close();
            }
            catch (JMSException e)
            {
            }
         }
      }
   }
```

In JBoss Application Server you can use the JMS JCA adapter for sending messages from EJBs (including Session, Entity and Message Driven Beans), Servlets (including jsps) and custom MBeans.

32.3. Configuring the JCA Adaptor

The Java Connector Architecture (JCA) Adapter is what allows HornetQ to be integrated with JEE components such as MDB's and EJB's. It configures how components such as MDB's consume messages from the HornetQ server and also how components such as EJB's or Servlet's can send messages.

The HornetQ JCA adapter is deployed via the `jms-ra.rar` archive. The configuration of the adapter is found in this archive under `META-INF/ra.xml`.

The configuration will look something like the following:

```
<resourceadapter>
      <resourceadapter-class>org.hornetq.ra.HornetQResourceAdapter</resourceadapter-class>
      <config-property>
         <description>The transport type</description>
         <config-property-name>ConnectorClassName</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
```

```
         <config-property-value>org.hornetq.core.remoting.impl.invm.InVMConnectorF
         actory</config-property-value>
      </config-property>
      <config-property>
         <description>The transport configuration. These values must be in the form of key=val;key=val;</
         <config-property-name>ConnectionParameters</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>server-id=0</config-property-value>
      </config-property>

      <outbound-resourceadapter>
         <connection-definition>
            <managedconnectionfactory-class>org.hornetq.ra.HornetQRAManagedConnection
            Factory</managedconnectionfactory-class>

            <config-property>
               <description>The default session type</description>
               <config-property-name>SessionDefaultType</config-property-name>
               <config-property-type>java.lang.String</config-property-type>
               <config-property-value>javax.jms.Queue</config-property-value>
            </config-property>
            <config-property>
               <description>Try to obtain a lock within specified number of seconds; less
               than or equal to 0 disable this functionality</description>
               <config-property-name>UseTryLock</config-property-name>
               <config-property-type>java.lang.Integer</config-property-type>
               <config-property-value>0</config-property-value>
            </config-property>

            <connectionfactory-interface>org.hornetq.ra.HornetQRAConnectionFactory
            </connectionfactory-interface>
            <connectionfactororg.hornetq.ra.HornetQConnectionFactoryImplonFactoryImpl
            </connectionfactory-impl-class>
            <connection-interface>javax.jms.Session</connection-interface>
            <connection-impl-class>org.hornetq.ra.HornetQRASession
            </connection-impl-class>
         </connection-definition>
         <transaction-support>XATransaction</transaction-support>
         <authentication-mechanism>
            <authentication-mechanism-type>BasicPassword
            </authentication-mechanism-type>
            <credential-interface>javax.resource.spi.security.PasswordCredential
            </credential-interface>
         </authentication-mechanism>
         <reauthentication-support>false</reauthentication-support>
      </outbound-resourceadapter>

      <inbound-resourceadapter>
         <messageadapter>
            <messagelistener>
               <messagelistener-type>javax.jms.MessageListener</messagelistener-type>
               <activationspec>
                  <activationspec-class>org.hornetq.ra.inflow.HornetQActivationSpec
                  </activationspec-class>
                  <required-config-property>
                      <config-property-name>destination</config-property-name>
                  </required-config-property>
               </activationspec>
            </messagelistener>
         </messageadapter>
      </inbound-resourceadapter>

   </resourceadapter>
```

There are 3 main parts to this configuration.

1. A set of global properties for the adapter
2. The configuration for the outbound part of the adapter. This is used for creating JMS resources within EE components.
3. The configuration of the inbound part of the adapter. This is used for controlling the consumption of messages via MDB's.

32.3.1. Adapter Global properties

The first element you see is `resourceadapter-class` which should be left unchanged. This is the HornetQ resource adapter class.

After that there is a list of configuration properties. This will be where most of the configuration is done. The first 2 configure the transport used by the adapter and the rest configure the connection factory itself.

Note

All connection factory properties will use the defaults when not provided. This is accept for the `reconnectAttempts` which will default to -1 which signifies that the connection should attempt to reconnect on connection failure indefinately. This is only used when the adapter is configured to connect to a remote server as an InVM connector can never fail.

The following table explains what each property is for.

Table 32.1. Global Configuration Properties

Property Name	Property Type	Property Description
ConnectorClassName	String	The Connector class name see Chapter 16 for info on available connectors
ConnectionParameters	String	The transport configuration. These values must be in the form of key=val;key=val; and will be specific to the connector used
useLocalTx	boolean	True will enable local transaction optimisation.
UseXA	boolean	Whether XA should be used
UserName	String	The user name to use when making a connection
Password	String	The password to use when making a connection
BackUpTransportType	String	The back up transport to use on failure.

Property Name	Property Type	Property Description
TransportConfiguration	String	The back up transport configuration
DiscoveryGroupAddress	String	The discovery group address to use to autodetect a server
DiscoveryGroupPort	integer	The port to use for discovery
DiscoveryRefreshTimeout	long	The timeout, in milli seconds, to refresh.
DiscoveryInitialWaitTimeout	long	The initial time to wait for discovery.
LoadBalancingPolicyClassName	String	The load balancing policy class to use.
PingPeriod	long	The period, in milliseconds, to ping the server for failure.
ConnectionTTL	long	The time to live for the connection.
CallTimeout	long	the call timeout, in milli seconds, for each packet sent.
DupsOKBatchSize	integer	The batch size of message acks to use if Dups ok is used.

continued..

TransactionBatchSize	integer	The batch size to use for sending messages within a transaction
ConsumerWindowSize	integer	The window size for the consumers internal buffer.
ConsumerMaxRate	integer	The max rate a consumer can receive.
ConfirmationWindowSize	integer	The window size for confirmations.
ProducerMaxRate	integer	The max rate a producer can send messages.
MinLargeMessageSize	integer	The size a message can be, in bytes, before it is sent as a multi part large message.
BlockOnAcknowledge	boolean	If true then block on acknowledge of messages.
BlockOnNonDurableSend	boolean	If true then block when sending

		non-durable messages
BlockOnDurableSend	boolean	If true then block when sending durable messages
AutoGroup	boolean	If true then auto group messages
PreAcknowledge	boolean	Whether to pre acknowledge messages before sending to consumer
reconnectAttempts	Integer	How attemts to try at reconnecting, default is -1
RetryInterval	long	How long to wait , in milli seconds, before retrying a failed connection
RetryIntervalMultiplier	double	Used for calculating the retry interval
FailoverOnServerShutdown	boolean	If true client will reconnect to another server if available
ClientID	String	The client ID of the connection

32.3.2. Adapter Outbound configuration

The outbound configuration should remain unchanged as they define connection factories that are used by Java EE components. These Connection Factories can be defined inside a configuration file that matches the name `*-ds.xml`. You'll find a default `jms-ds.xml` configuration under the `hornetq.sar` directory in the Jboss AS deployment. The connection factories defined in the config file inherit their properties from the main `ra.xml` configuration but can also be overridden, the following example show how to define one.

Please note that this configuration only applies to install the HornetQ resource adapter in the JBoss Application Server. If you are using another JEE application server please refer to your application servers documentation for how to do this.

```
<tx-connection-factory>
      <jndi-name>RemoteJmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-definition>org.hornetq.ra.HornetQRAConnectionFactory
</connection-definition>
      <config-property name="SessionDefaultType" type="String">javax.jms.Topic
      </config-property>
      <config-property name="ConnectorClassName" type="String">
        org.hornetq.integration.transports.netty.NettyConnectorFactory
      </config-property>
      <config-property name="ConnectionParameters" type="String">
          port=5445</config-property>
      <max-pool-size>20</max-pool-size>
</tx-connection-factory>
```

In this example the connection factory will be bound to JNDI with the name `RemoteJmsXA` and can be looked up in

the usual way using JNDI or defined within the EJB or MDB as such:

```
@Resource(mappedName="java:RemoteJmsXA")
private ConnectionFactory connectionFactory;
```

The `config-property` elements are what over rides those in the `ra.xml` config. Any of the elements pertaining to the connection factory can be over ridden here.

32.3.3. Adapter Inbound configuration

The inbound configuration should again remain unchanged. This controls what forwards messages onto MDB's. It is possible to override properties on the MDB by adding an activation configuration to the MDB itself. This could be used to configure the MDB to consume from a different server. The next section demonstrates over riding the configuration.

32.4. High Availability JNDI (HA-JNDI)

If you are using JNDI to look-up JMS queues, topics and connection factories from a cluster of servers, it is likely you will want to use HA-JNDI so that your JNDI look-ups will continue to work if one or more of the servers in the cluster fail.

HA-JNDI is a JBoss Application Server service which allows you to use JNDI from clients without them having to know the exact JNDI connection details of every server in the cluster. This service is only available if using a cluster of JBoss Application Server instances.

To use it use the following properties when connecting to JNDI.

```
Hashtable<String, String> jndiParameters = new Hashtable<String, String>();
jndiParameters.put("java.naming.factory.initial",
    "org.jnp.interfaces.NamingContextFactory");
jndiParameters.put("java.naming.factory.url.pkgs=",
    "org.jboss.naming:org.jnp.interfaces");

initialContext = new InitialContext(jndiParameters);
```

For more information on using HA-JNDI see the JBoss Application Server clustering documentation [http://www.jboss.org/file-access/default/members/jbossas/freezone/docs/Clustering_Guide/5/html/clustering-jndi.html]

32.5. XA Recovery

XA recovery deals with system or application failures to ensure that of a transaction are applied consistently to all resources affected by the transaction, even if any of the application processes or the machine hosting them crash or lose network connectivity. For more information on XA Recovery,please refer to JBoss Transactions [http://www.jboss.org/community/wiki/JBossTransactions].

When HornetQ is integrated with JBoss AS, it can take advantage of JBoss Transactions to provide recovery of messaging resources. If messages are involved in a XA transaction, in the event of a server crash, the recovery manager will ensure that the transactions are recovered and the messages will either be committed or rolled back

(depending on the transaction outcome) when the server is restarted.

32.5.1. XA Recovery Configuration

To enable HornetQs XA Recovery, the Recovery Manager must be configured to connect to HornetQ to recover its resources. The following property must be added to the `jta` section of `conf/jbossts-properties.xml` of JBoss AS profiles:

```
<properties depends="arjuna" name="jta">
   ...

   <property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HornetQ1"
             value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;[connection configuratio
</properties>
```

The `[connection configuration]` contains all the information required to connect to HornetQ node under the form `[connector factory class name],[user name], [password], [connector parameters]`.

- `[connector factory class name]` corresponds to the name of the `ConnectorFactory` used to connect to HornetQ. Values can be `org.hornetq.core.remoting.impl.invm.InVMConnectorFactory` or `org.hornetq.integration.transports.netty.NettyConnectorFactory`
- `[user name]` is the user name to create a client session. It is optional
- `[password]` is the password to create a client session. It is mandatory only if the user name is specified
- `[connector parameters]` is a list of comma-separated key=value pair which are passed to the connector factory (see Chapter 16 for a list of the transport parameters).

Note

HornetQ must have a valid acceptor which corresponds to the connector specified in `conf/jbossts-properties.xml`.

32.5.1.1. Configuration Settings

If HornetQ is configured with a default in-vm acceptor:

```
<acceptor name="in-vm">
    <factory-class>org.hornetq.core.remoting.impl.invm.InVMAcceptorFactory</factory-class>
</acceptor>
```

the corresponding configuration in `conf/jbossts-properties.xml` is:

```
<property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"
   value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;org.hornetq.core.remoting.impl.invm.I
```

If it is now configured with a netty acceptor on a non-default port:

```
<acceptor name="netty">
    <factory-class>org.hornetq.integration.transports.netty.NettyAcceptorFactory</factory-class>
    <param key="port" value="8888"/>
</acceptor>
```

the corresponding configuration in `conf/jbossts-properties.xml` is:

```
<property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"
       value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;org.hornetq.integration.transport
```

Note

Note the additional commas to skip the user and password before connector parameters

If the recovery must use `admin, adminpass`, the configuration would have been:

```
                    <property name="com.arjuna.ats.jta.recovery.XAResourceRecovery.HORNETQ1"
                          value="org.hornetq.jms.server.recovery.HornetQXAResourceRecovery;org.hornetq.i
```

Configuring HornetQ with an invm acceptor and configuring the Recovery Manager with an invm connector is the recommended way to enable XA Recovery.

32.5.2. Example

See Section 11.3.8 which shows how to configure XA Recovery and recover messages after a server crash.

33

The JMS Bridge

HornetQ includes a fully functional JMS message bridge.

The function of the bridge is to consume messages from a source queue or topic, and send them to a target queue or topic, typically on a different server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, and where the connection may be unreliable.

A bridge can be deployed as a standalone application, with HornetQ standalone server or inside a JBoss AS instance. The source and the target can be located in the same virtual machine or another one.

The bridge can also be used to bridge messages from other non HornetQ JMS servers, as long as they are JMS 1.1 compliant.

Note

Do not confuse a JMS bridge with a core bridge. A JMS bridge can be used to bridge any two JMS 1.1 compliant JMS providers and uses the JMS API. A core bridge (described in Chapter 36) is used to bridge any two HornetQ instances and uses the core API. Always use a core bridge if you can in preference to a JMS bridge. The core bridge will typically provide better performance than a JMS bridge. Also the core bridge can provide *once and only once* delivery guarantees without using XA.

The bridge has built-in resilience to failure so if the source or target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the source and/or target until they come back online. When it comes back online it will resume operation as normal.

The bridge can be configured with an optional JMS selector, so it will only consume messages matching that JMS selector

It can be configured to consume from a queue or a topic. When it consumes from a topic it can be configured to consume using a non durable or durable subscription

Typically, the bridge is deployed by the JBoss Micro Container via a beans configuration file. This would typically be deployed inside the JBoss Application Server and the following example shows an example of a beans file that bridges 2 destinations which are actually on the same server.

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

       <bean name="JMSBridge" class="org.hornetq.api.jms.bridge.impl.JMSBridgeImpl">
           <!-- HornetQ must be started before the bridge -->
           <depends>HornetQServer</depends>
           <constructor>
```

```
            <!-- Source ConnectionFactory Factory -->
            <parameter>
                <inject bean="SourceCFF"/>
            </parameter>
            <!-- Target ConnectionFactory Factory -->
            <parameter>
                <inject bean="TargetCFF"/>
            </parameter>
            <!-- Source DestinationFactory -->
            <parameter>
                <inject bean="SourceDestinationFactory"/>
            </parameter>
            <!-- Target DestinationFactory -->
            <parameter>
                <inject bean="TargetDestinationFactory"/>
            </parameter>
            <!-- Source User Name (no username here) -->
            <parameter><null /></parameter>
            <!-- Source Password (no password here)-->
            <parameter><null /></parameter>
            <!-- Target User Name (no username here)-->
            <parameter><null /></parameter>
            <!-- Target Password (no password here)-->
            <parameter><null /></parameter>
            <!-- Selector -->
            <parameter><null /></parameter>
            <!-- Failure Retry Interval (in ms) -->
            <parameter>5000</parameter>
            <!-- Max Retries -->
            <parameter>10</parameter>
            <!-- Quality Of Service -->
            <parameter>ONCE_AND_ONLY_ONCE</parameter>
            <!-- Max Batch Size -->
            <parameter>1</parameter>
            <!-- Max Batch Time (-1 means infinite) -->
            <parameter>-1</parameter>
            <!-- Subscription name (no subscription name here)-->
            <parameter><null /></parameter>
            <!-- Client ID  (no client ID here)-->
            <parameter><null /></parameter>
            <!-- Add MessageID In Header -->
            <parameter>true</parameter>
            <!-- register the JMS Bridge in the AS MBeanServer -->
            <parameter>
                <inject bean="MBeanServer"/>
            </parameter>
            <parameter>org.hornetq:service=JMSBridge</parameter>
          </constructor>
        <property name="transactionManager">
            <inject bean="RealTransactionManager"/>
        </property>
    </bean>

    <!-- SourceCFF describes the ConnectionFactory used to connect to the
         source destination -->
    <bean name="SourceCFF"
         class="org.hornetq.api.jms.bridge.impl.JNDIConnectionFactoryFactory">
        <constructor>
            <parameter>
                <inject bean="JNDI" />
            </parameter>
            <parameter>/ConnectionFactory</parameter>
        </constructor>
    </bean>

    <!-- TargetCFF describes the ConnectionFactory used to connect to the
```

```
        target destination -->
       <bean name="TargetCFF"
            class="org.hornetq.api.jms.bridge.impl.JNDIConnectionFactoryFactory">
           <constructor>
               <parameter>
                   <inject bean="JNDI" />
               </parameter>
               <parameter>/ConnectionFactory</parameter>
           </constructor>
       </bean>

       <!-- SourceDestinationFactory describes the Destination used as the source -->
       <bean name="SourceDestinationFactory"
            class="org.hornetq.api.jms.bridge.impl.JNDIDestinationFactory">
           <constructor>
               <parameter>
                   <inject bean="JNDI" />
               </parameter>
               <parameter>/queue/source</parameter>
           </constructor>
       </bean>

       <!-- TargetDestinationFactory describes the Destination used as the target -->
       <bean name="TargetDestinationFactory"
            class="org.hornetq.api.jms.bridge.impl.JNDIDestinationFactory">
           <constructor>
               <parameter>
                   <inject bean="JNDI" />
               </parameter>
               <parameter>/queue/target</parameter>
           </constructor>
       </bean>

       <!-- JNDI is a Hashtable containing the JNDI properties required -->
       <!-- to connect to the sources and targets JMS resrouces         -->
      <bean name="JNDI" class="java.util.Hashtable">
         <constructor class="java.util.Map">
            <map class="java.util.Hashtable" keyClass="String"
                                             valueClass="String">
               <entry>
                  <key>java.naming.factory.initial</key>
                  <value>org.jnp.interfaces.NamingContextFactory</value>
               </entry>
               <entry>
                  <key>java.naming.provider.url</key>
                  <value>jnp://localhost:1099</value>
               </entry>
               <entry>
                  <key>java.naming.factory.url.pkgs</key>
                  <value>org.jboss.naming:org.jnp.interfaces"</value>
               </entry>
            </map>
         </constructor>
      </bean>

      <bean name="MBeanServer" class="javax.management.MBeanServer">
         <constructor factoryClass="org.jboss.mx.util.MBeanServerLocator"
                      factoryMethod="locateJBoss"/>
      </bean>
</deployment>
```

33.1. JMS Bridge Parameters

The main bean deployed is the `JMSBridge` bean. The bean is configurable by the parameters passed to its constructor.

Note

To let a parameter be unspecified (for example, if the authentication is anonymous or no message selector is provided), use `<null />` for the unspecified parameter value.

- Source Connection Factory Factory

  This injects the `SourceCFF` bean (also defined in the beans file). This bean is used to create the *source* `ConnectionFactory`

- Target Connection Factory Factory

  This injects the `TargetCFF` bean (also defined in the beans file). This bean is used to create the *target* `ConnectionFactory`

- Source Destination Factory Factory

  This injects the `SourceDestinationFactory` bean (also defined in the beans file). This bean is used to create the *source* `Destination`

- Target Destination Factory Factory

  This injects the `TargetDestinationFactory` bean (also defined in the beans file). This bean is used to create the *target* `Destination`

- Source User Name

  this parameter is the username for creating the *source* connection

- Source Password

  this parameter is the parameter for creating the *source* connection

- Target User Name

  this parameter is the username for creating the *target* connection

- Target Password

  this parameter is the password for creating the *target* connection

- Selector

  This represents a JMS selector expression used for consuming messages from the source destination. Only messages that match the selector expression will be bridged from the source to the target destination

  The selector expression must follow the JMS selector syntax [http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html]

- Failure Retry Interval

  This represents the amount of time in ms to wait between trying to recreate connections to the source or target servers when the bridge has detected they have failed

- Max Retries

  This represents the number of times to attempt to recreate connections to the source or target servers when the bridge has detected they have failed. The bridge will give up after trying this number of times. `-1` represents 'try forever'

- Quality Of Service

  This parameter represents the desired quality of service mode

  Possible values are:

  - `AT_MOST_ONCE`
  - `DUPLICATES_OK`
  - `ONCE_AND_ONLY_ONCE`

  See Section 33.4 for a explanation of these modes.

- Max Batch Size

  This represents the maximum number of messages to consume from the source destination before sending them in a batch to the target destination. Its value must `>= 1`

- Max Batch Time

  This represents the maximum number of milliseconds to wait before sending a batch to target, even if the number of messages consumed has not reached `MaxBatchSize`. Its value must be `-1` to represent 'wait forever', or `>= 1` to specify an actual time

- Subscription Name

  If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this parameter represents the durable subscription name

- Client ID

  If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this attribute represents the the JMS client ID to use when creating/looking up the durable subscription

- Add MessageID In Header

  If `true`, then the original message's message ID will be appended in the message sent to the destination in the header `HORNETQ_BRIDGE_MSG_ID_LIST`. If the message is bridged more than once, each message ID will be appended. This enables a distributed request-response pattern to be used

Note

when you receive the message you can send back a response using the correlation id of the first message id, so when the original sender gets it back it will be able to correlate it.

- MBean Server

  To manage the JMS Bridge using JMX, set the MBeanServer where the JMS Bridge MBean must be registered (e.g. the JVM Platform MBeanServer or JBoss AS MBeanServer)

- ObjectName

  If you set the MBeanServer, you also need to set the ObjectName used to register the JMS Bridge MBean (must be unique)

33.2. Source and Target Connection Factories

The source and target connection factory factories are used to create the connection factory used to create the connection for the source or target server.

The configuration example above uses the default implementation provided by HornetQ that looks up the connection factory using JNDI. For other Application Servers or JMS providers a new implementation may have to be provided. This can easily be done by implementing the interface `org.hornetq.jms.bridge.ConnectionFactoryFactory`.

33.3. Source and Target Destination Factories

Again, similarly, these are used to create or lookup up the destinations.

In the configuration example above, we have used the default provided by HornetQ that looks up the destination using JNDI.

A new implementation can be provided by implementing `org.hornetq.jms.bridge.DestinationFactory` interface.

33.4. Quality Of Service

The quality of service modes used by the bridge are described here in more detail.

33.4.1. AT_MOST_ONCE

With this QoS mode messages will reach the destination from the source at most once. The messages are consumed from the source and acknowledged before sending to the destination. Therefore there is a possibility that if failure occurs between removing them from the source and them arriving at the destination they could be lost. Hence delivery will occur at most once.

This mode is available for both durable and non-durable messages.

33.4.2. DUPLICATES_OK

With this QoS mode, the messages are consumed from the source and then acknowledged after they have been successfully sent to the destination. Therefore there is a possibility that if failure occurs after sending to the destination but before acknowledging them, they could be sent again when the system recovers. I.e. the destination might receive duplicates after a failure.

This mode is available for both durable and non-durable messages.

33.4.3. ONCE_AND_ONLY_ONCE

This QoS mode ensures messages will reach the destination from the source once and only once. (Sometimes this mode is known as "exactly once"). If both the source and the destination are on the same HornetQ server instance then this can be achieved by sending and acknowledging the messages in the same local transaction. If the source and destination are on different servers this is achieved by enlisting the sending and consuming sessions in a JTA transaction. The JTA transaction is controlled by JBoss Transactions JTA * implementation which is a fully recovering transaction manager, thus providing a very high degree of durability. If JTA is required then both supplied connection factories need to be XAConnectionFactory implementations. This is likely to be the slowest mode since it requires extra persistence for the transaction logging.

This mode is only available for durable messages.

Note

For a specific application it may possible to provide once and only once semantics without using the ONCE_AND_ONLY_ONCE QoS level. This can be done by using the DUPLICATES_OK mode and then checking for duplicates at the destination and discarding them. Some JMS servers provide automatic duplicate message detection functionality, or this may be possible to implement on the application level by maintaining a cache of received message ids on disk and comparing received messages to them. The cache would only be valid for a certain period of time so this approach is not as watertight as using ONCE_AND_ONLY_ONCE but may be a good choice depending on your specific application.

33.4.4. Examples

Please see Section 11.3.4 which shows how to configure and use a JMS Bridge with JBoss AS to send messages to the source destination and consume them from the target destination.

Please see Section 11.1.20 which shows how to configure and use a JMS Bridge between two standalone HornetQ servers.

34

Client Reconnection and Session Reattachment

HornetQ clients can be configured to automatically reconnect or re-attach to the server in the event that a failure is detected in the connection between the client and the server.

34.1. 100% Transparent session re-attachment

If the failure was due to some transient failure such as a temporary network failure, and the target server was not restarted, then the sessions will still be existent on the server, asssuming the client hasn't been disconnected for more than connection-ttl Chapter 17.

In this scenario, HornetQ will automatically re-attach the client sessions to the server sessions when the connection reconnects. This is done 100% transparently and the client can continue exactly as if nothing had happened.

The way this works is as follows:

As HornetQ clients send commands to their servers they store each sent command in an in-memory buffer. In the case that connection failure occurs and the client subsequently reattaches to the same server, as part of the reattachment protocol the server informs the client during reattachment with the id of the last command it successfully received from that client.

If the client has sent more commands than were received before failover it can replay any sent commands from its buffer so that the client and server can reconcile their states.

The size of this buffer is configured by the `ConfirmationWindowSize` parameter, when the server has received `ConfirmationWindowSize` bytes of commands and processed them it will send back a command confirmation to the client, and the client can then free up space in the buffer.

If you are using JMS and you're using the JMS service on the server to load your JMS connection factory instances into JNDI then this parameter can be configured in `hornetq-jms.xml` using the element `confirmation-window-size` a. If you're using JMS but not using JNDI then you can set these values directly on the `HornetQConnectionFactory` instance using the appropriate setter method.

If you're using core you can set these values directly on the `ClientSessionFactory` instance using the appropriate setter method.

The window is specified in bytes, and has a default value of `1MiB`.

Setting this parameter to `-1` disables any buffering and prevents any re-attachment from occurring, forcing reconnect instead. The default value for this parameter is `-1`.

34.2. Session reconnection

Alternatively, the server might have actually been restarted after crashing or being stopped. In this case any sessions will no longer be existent on the server and it won't be possible to 100% transparently re-attach to them.

In this case, HornetQ will automatically reconnect the connection and *recreate* any sessions and consumers on the server corresponding to the sessions and consumers on the client. This process is exactly the same as what happens during failover onto a backup server.

Client reconnection is also used internally by components such as core bridges to allow them to reconnect to their target servers.

Please see the section on failover Section 39.2.1 to get a full understanding of how transacted and non-transacted sessions are reconnected during failover/reconnect and what you need to do to maintain *once and only once* delivery guarantees.

34.3. Configuring reconnection/reattachment attributes

Client reconnection is configured using the following parameters:

- `retry-interval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `2000` milliseconds.
- `retry-interval-multiplier`. This optional parameter determines determines a multiplier to apply to the time since the last retry to compute the time to the next retry.

  This allows you to implement an *exponential backoff* between retry attempts.

  Let's take an example:

  If we set `retry-interval` to `1000` ms and we set `retry-interval-multiplier` to `2.0`, then, if the first reconnect attempt fails, we will wait `1000` ms then `2000` ms then `4000` ms between subsequent reconnection attempts.

  The default value is `1.0` meaning each reconnect attempt is spaced at equal intervals.
- `max-retry-interval`. This optional parameter determines the maximum retry interval that will be used. When setting `retry-interval-multiplier` it would otherwise be possible that subsequent retries exponentially increase to ridiculously large values. By setting this parameter you can set an upper limit on that value. The default value is `2000` milliseconds.
- `reconnect-attempts`. This optional parameter determines the total number of reconnect attempts to make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `0`.

If you're using JMS, and you're using the JMS Service on the server to load your JMS connection factory instances directly into JNDI, then you can specify these parameters in the xml configuration in `hornetq-jms.xml`, for example:

```
<connection-factory name="ConnectionFactory">
<connectors>
   <connector-ref connector-name="netty"/>
</connectors>
<entries>
   <entry name="ConnectionFactory"/>
   <entry name="XAConnectionFactory"/>
</entries>
<retry-interval>1000</retry-interval>
<retry-interval-multiplier>1.5</retry-interval-multiplier>
<max-retry-interval>60000</max-retry-interval>
<reconnect-attempts>1000</reconnect-attempts>
</connection-factory>
```

If you're using JMS, but instantiating your JMS connection factory directly, you can specify the parameters using the appropriate setter methods on the `HornetQConnectionFactory` immediately after creating it.

If you're using the core API and instantiating the `ClientSessionFactory` instance directly you can also specify the parameters using the appropriate setter methods on the `ClientSessionFactory` immediately after creating it.

If your client does manage to reconnect but the session is no longer available on the server, for instance if the server has been restarted or it has timed out, then the client won't be able to re-attach, and any `ExceptionListener` or `FailureListener` instances registered on the connection or session will be called.

34.4. ExceptionListeners and SessionFailureListeners

Please note, that when a client reconnects or re-attaches, any registered JMS `ExceptionListener` or core API `SessionFailureListener` will be called.

35

Diverting and Splitting Message Flows

HornetQ allows you to configure objects called *diverts* with some simple server configuration.

Diverts allow you to transparently divert messages routed to one address to some other address, without making any changes to any client application logic.

Diverts can be *exclusive*, meaning that that the message is diverted to the new address, and does not go to the old address at all, or they can be *non-exclusive* which means the message continues to go the old address, and a *copy* of it is also sent to the new address. Non-exclusive diverts can therefore be used for *splitting* message flows, e.g. there may be a requirement to monitor every order sent to an order queue.

Diverts can also be configured to have an optional message filter. If specified then only messages that match the filter will be diverted.

Diverts can also be configured to apply a `Transformer`. If specified, all diverted messages will have the opportunity of being transformed by the `Transformer`.

A divert will only divert a message to an address on the *same server*, however, if you want to divert to an address on a different server, a common pattern would be to divert to a local store-and-forward queue, then set up a bridge which consumes from that queue and forwards to an address on a different server.

Diverts are therefore a very sophisticated concept, which when combined with bridges can be used to create interesting and complex routings. The set of diverts on a server can be thought of as a type of routing table for messages. Combining diverts with bridges allows you to create a distributed network of reliable routing connections between multiple geographically distributed servers, creating your global messaging mesh.

Diverts are defined as xml in the `hornetq-configuration.xml` file. There can be zero or more diverts in the file.

Please see Section 11.1.13 for a full working example showing you how to configure and use diverts.

Let's take a look at some divert examples:

35.1. Exclusive Divert

Let's take a look at an exclusive divert. An exclusive divert diverts all matching messages that are routed to the old address to the new address. Matching messages do not get routed to the old address.

Here's some example xml configuration for an exclusive divert, it's taken from the divert example:

```
<divert name="prices-divert">
    <address>jms.topic.priceUpdates</address>
    <forwarding-address>jms.queue.priceForwarding</forwarding-address>
```

```
    <filter string="office='New York'"/>
    <transformer-class-name>
        org.hornetq.jms.example.AddForwardingTimeTransformer
    </transformer-class-name>
    <exclusive>true</exclusive>
</divert>
```

We define a divert called '`prices-divert`' that will divert any messages sent to the address '`jms.topic.priceUpdates`' (this corresponds to any messages sent to a JMS Topic called '`priceUpdates`') to another local address '`jms.queue.priceForwarding`' (this corresponds to a local JMS queue called '`priceForwarding`'

We also specify a message filter string so only messages with the message property `office` with value `New York` will get diverted, all other messages will continue to be routed to the normal address. The filter string is optional, if not specified then all messages will be considered matched.

In this example a transformer class is specified. Again this is optional, and if specified the transformer will be executed for each matching message. This allows you to change the messages body or properties before it is diverted. In this example the transformer simply adds a header that records the time the divert happened.

This example is actually diverting messages to a local store and forward queue, which is configured with a bridge which forwards the message to an address on another HornetQ server. Please see the example for more details.

35.2. Non-exclusive Divert

Now we'll take a look at a non-exclusive divert. Non exclusive diverts are the same as exclusive diverts, but they only forward a *copy* of the message to the new address. The original message continues to the old address

You can therefore think of non-exclusive diverts as *splitting* a message flow.

Non exclusive diverts can be configured in the same was as exclusive diverts with an optional filter and transformer, here's an example non-exclusive divert, again from the divert example:

```
<divert name="order-divert">
    <address>jms.queue.orders</address>
    <forwarding-address>jms.topic.spyTopic</forwarding-address>
    <exclusive>false</exclusive>
</divert>
```

The above divert example takes a copy of every message sent to the address '`jms.queue.orders`' (Which corresponds to a JMS Queue called '`orders`') and sends it to a local address called '`jms.topic.SpyTopic`' (which corresponds to a JMS Topic called '`SpyTopic`').

36

Core Bridges

The function of a bridge is to consume messages from a source queue, and forward them to a target address, typically on a different HornetQ server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, or internet and where the connection may be unreliable.

The bridge has built in resilience to failure so if the target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

In summary, bridges are a way to reliably connect two separate HornetQ servers together. With a core bridge both source and target servers must be HornetQ servers.

Bridges can be configured to provide *once and only once* delivery guarantees even in the event of the failure of the source or the target server. They do this by using duplicate detection (described in Chapter 37).

Note

Although they have similar function, don't confuse core bridges with JMS bridges!

Core bridges are for linking a HornetQ node with another HornetQ node and do not use the JMS API. A JMS Bridge is used for linking any two JMS 1.1 compliant JMS providers. So, a JMS Bridge could be used for bridging to or from different JMS compliant messaging system. It's always preferable to use a core bridge if you can. Core bridges use duplicate detection to provide *once and only once* guarantees. To provide the same guarantee using a JMS bridge you would have to use XA which has a higher overhead and is more complex to configure.

36.1. Configuring Bridges

Bridges are configured in `hornetq-configuration.xml`. Let's kick off with an example (this is actually from the bridge example):

```
<bridge name="my-bridge">
    <queue-name>jms.queue.sausage-factory</queue-name>
    <forwarding-address>jms.queue.mincing-machine</forwarding-address>
    <filter-string="name='aardvark'"/>
    <transformer-class-name>
        org.hornetq.jms.example.HatColourChangeTransformer
    </transformer-class-name>
    <retry-interval>1000</retry-interval>
    <retry-interval-multiplier>1.0</retry-interval-multiplier>
```

```
    <reconnect-attempts>-1</reconnect-attempts>
    <failover-on-server-shutdown>false</failover-on-server-shutdown>
    <use-duplicate-detection>true</use-duplicate-detection>
    <confirmation-window-size>10000000</confirmation-window-size>
    <connector-ref connector-name="remote-connector"
        backup-connector-name="backup-remote-connector"/>
    <user>foouser</user>
    <password>foopassword</password>
</bridge>
```

In the above example we have shown all the parameters its possible to configure for a bridge. In practice you might use many of the defaults so it won't be necessary to specify them all explicitly.

Let's take a look at all the parameters in turn:

- `name` attribute. All bridges must have a unique name in the server.
- `queue-name`. This is the unique name of the local queue that the bridge consumes from, it's a mandatory parameter.

  The queue must already exist by the time the bridge is instantiated at start-up.

  ### Note

  If you're using JMS then normally the JMS configuration `hornetq-jms.xml` is loaded after the core configuration file `hornetq-configuration.xml` is loaded. If your bridge is consuming from a JMS queue then you'll need to make sure the JMS queue is also deployed as a core queue in the core configuration. Take a look at the bridge example for an example of how this is done.

- `forwarding-address`. This is the address on the target server that the message will be forwarded to. If a forwarding address is not specified, then the original address of the message will be retained.
- `filter-string`. An optional filter string can be supplied. If specified then only messages which match the filter expression specified in the filter string will be forwarded. The filter string follows the HornetQ filter expression syntax described in Chapter 14.
- `transformer-class-name`. An optional transformer-class-name can be specified. This is the name of a user-defined class which implements the `org.hornetq.core.server.cluster.Transformer` interface.

  If this is specified then the transformer's `transform()` method will be invoked with the message before it is forwarded. This gives you the opportunity to transform the message's header or body before forwarding it.
- `retry-interval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `2000`milliseconds.
- `retry-interval-multiplier`. This optional parameter determines determines a multiplier to apply to the time since the last retry to compute the time to the next retry.

  This allows you to implement an *exponential backoff* between retry attempts.

  Let's take an example:

If we set `retry-interval`to `1000` ms and we set `retry-interval-multiplier` to `2.0`, then, if the first reconnect attempt fails, we will wait `1000` ms then `2000` ms then `4000` ms between subsequent reconnection attempts.

The default value is `1.0` meaning each reconnect attempt is spaced at equal intervals.

- `reconnect-attempts`. This optional parameter determines the total number of reconnect attempts the bridge will make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `-1`.

- `failover-on-server-shutdown`. This optional parameter determines whether the bridge will attempt to failover onto a backup server (if specified) when the target server is cleanly shutdown rather than crashed.

  The bridge connector can specify both a live and a backup server, if it specifies a backup server and this parameter is set to `true` then if the target server is *cleanly* shutdown the bridge connection will attempt to failover onto its backup. If the bridge connector has no backup server configured then this parameter has no effect.

  Sometimes you want a bridge configured with a live and a backup target server, but you don't want to failover to the backup if the live server is simply taken down temporarily for maintenance, this is when this parameter comes in handy.

  The default value for this parameter is `false`.

- `use-duplicate-detection`. This optional parameter determines whether the bridge will automatically insert a duplicate id property into each message that it forwards.

  Doing so, allows the target server to perform duplicate detection on messages it receives from the source server. If the connection fails or server crashes, then, when the bridge resumes it will resend unacknowledged messages. This might result in duplicate messages being sent to the target server. By enabling duplicate detection allows these duplicates to be screened out and ignored.

  This allows the bridge to provide a *once and only once* delivery guarantee without using heavyweight methods such as XA (see Chapter 37 for more information).

  The default value for this parameter is `true`.

- `confirmation-window-size`. This optional parameter determines the `confirmation-window-size` to use for the connection used to forward messages to the target node. This attribute is described in section Chapter 34

  ## Warning

  When using the bridge to forward messages from a queue which has a max-size-bytes set it's important that confirmation-window-size is less than or equal to `max-size-bytes` to prevent the flow of messages from ceasing.

- `connector-ref`. This mandatory parameter determines which *connector* pair the bridge will use to actually make the connection to the target server.

  A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Chapter 16.

The `connector-ref` element can be configured with two attributes:

- `connector-name`. This references the name of a connector defined in the core configuration file `hornetq-configuration.xml`. The bridge will use this connector to make its connection to the target server. This attribute is mandatory.
- `backup-connector-name`. This optional parameter also references the name of a connector defined in the core configuration file `hornetq-configuration.xml`. It represents the connector that the bridge will failover onto if it detects the live server connection has failed. If this is specified and `failover-on-server-shutdown` is set to `true` then it will also attempt failover onto this connector if the live target server is cleanly shut-down.

- `user`. This optional parameter determines the user name to use when creating the bridge connection to the remote server. If it is not specified the default cluster user specified by `cluster-user` in `hornetq-configuration.xml` will be used.
- `password`. This optional parameter determines the password to use when creating the bridge connection to the remote server. If it is not specified the default cluster password specified by `cluster-password` in `hornetq-configuration.xml` will be used.

37

Duplicate Message Detection

HornetQ includes powerful automatic duplicate message detection, filtering out duplicate messages without you having to code your own fiddly duplicate detection logic at the application level. This chapter will explain what duplicate detection is, how HornetQ uses it and how and where to configure it.

When sending messages from a client to a server, or indeed from a server to another server, if the target server or connection fails sometime after sending the message, but before the sender receives a response that the send (or commit) was processed successfully then the sender cannot know for sure if the message was sent successfully to the address.

If the target server or connection failed after the send was received and processed but before the response was sent back then the message will have been sent to the address successfully, but if the target server or connection failed before the send was received and finished processing then it will not have been sent to the address successfully. From the senders point of view it's not possible to distinguish these two cases.

When the server recovers this leaves the client in a difficult situation. It knows the target server failed, but it does not know if the last message reached its destination ok. If it decides to resend the last message, then that could result in a duplicate message being sent to the address. If each message was an order or a trade then this could result in the order being fulfilled twice or the trade being double booked. This is clearly not a desirable situation.

Sending the message(s) in a transaction does not help out either. If the server or connection fails while the transaction commit is being processed it is also indeterminate whether the transaction was successfully committed or not!

To solve these issues HornetQ provides automatic duplicate messages detection for messages sent to addresses.

37.1. Using Duplicate Detection for Message Sending

Enabling duplicate message detection for sent messages is simple: you just need to set a special property on the message to a unique value. You can create the value however you like, as long as it is unique. When the target server receives the message it will check if that property is set, if it is, then it will check in its in memory cache if it has already received a message with that value of the header. If it has received a message with the same value before then it will ignore the message.

Note

Using duplicate detection to move messages between nodes can give you the same *once and only once* delivery guarantees as if you were using an XA transaction to consume messages from source and send them to the target, but with less overhead and much easier configuration than using XA.

If you're sending messages in a transaction then you don't have to set the property for *every* message you send in that transaction, you only need to set it once in the transaction. If the server detects a duplicate message for any

message in the transaction, then it will ignore the entire transaction.

The name of the property that you set is given by the value of `org.hornetq.core.message.impl.HDR_DUPLICATE_DETECTION_ID`, which is `_HQ_DUPL_ID`

The value of the property can be of type `byte[]` or `SimpleString` if you're using the core API. If you're using JMS it must be a `String`, and its value should be unique. An easy way of generating a unique id is by generating a UUID.

Here's an example of setting the property using the core API:

```
...
ClientMessage message = session.createMessage(true);
SimpleString myUniqueID = "This is my unique id";   // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID, myUniqueID);
...
```

And here's an example using the JMS API:

```
...
Message jmsMessage = session.createMessage();
String myUniqueID = "This is my unique id";   // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
...
```

37.2. Configuring the Duplicate ID Cache

The server maintains caches of received values of the `org.hornetq.core.message.impl.HDR_DUPLICATE_DETECTION_ID` property sent to each address. Each address has its own distinct cache.

The cache is a circular fixed size cache. If the cache has a maximum size of `n` elements, then the `n + 1`th id stored will overwrite the `0`th element in the cache.

The maximum size of the cache is configured by the parameter `id-cache-size` in `hornetq-configuration.xml`, the default value is `2000` elements.

The caches can also be configured to persist to disk or not. This is configured by the parameter `persist-id-cache`, also in `hornetq-configuration.xml`. If this is set to `true` then each id will be persisted to permanent storage as they are received. The default value for this parameter is `true`.

Note

When choosing a size of the duplicate id cache be sure to set it to a larger enough size so if you resend messages all the previously sent ones are in the cache not having been overwritten.

37.3. Duplicate Detection and Bridges

Core bridges can be configured to automatically add a unique duplicate id value (if there isn't already one in the message) before forwarding the message to it's target. This ensures that if the target server crashes or the connection is interrupted and the bridge resends the message, then if it has already been received by the target server, it will be ignored.

To configure a core bridge to add the duplicate id header, simply set the *use-duplicate-detection* to `true` when configuring a bridge in `hornetq-configuration.xml`.

The default value for this parameter is `true`.

For more information on core bridges and how to configure them, please see Chapter 36.

37.4. Duplicate Detection and Cluster Connections

Cluster connections internally use core bridges to move messages reliable between nodes of the cluster. Consequently they can also be configured to insert the duplicate id header for each message they move using their internal bridges.

To configure a cluster connection to add the duplicate id header, simply set the *use-duplicate-detection* to `true` when configuring a cluster connection in `hornetq-configuration.xml`.

The default value for this parameter is `true`.

For more information on cluster connections and how to configure them, please see Chapter 38.

37.5. Duplicate Detection and Paging

HornetQ also uses duplicate detection when paging messages to storage. This is so when a message is depaged from storage and server failure occurs, we do not end up depaging the message more than once which could result in duplicate delivery.

For more information on paging and how to configure it, please see Chapter 24.

38

Clusters

38.1. Clusters Overview

HornetQ clusters allow groups of HornetQ servers to be grouped together in order to share message processing load. Each active node in the cluster is an active HornetQ server which manages its own messages and handles its own connections. A server must be configured to be clustered, you will need to set the `clustered` element in the `hornetq-configuration.xml` configuration file to `true`, this is `false` by default.

The cluster is formed by each node declaring *cluster connections* to other nodes in the core configuration file `hornetq-configuration.xml`. When a node forms a cluster connection to another node, internally it creates a *core bridge* (as described in Chapter 36) connection between it and the other node, this is done transparently behind the scenes - you don't have to declare an explicit bridge for each node. These cluster connections allow messages to flow between the nodes of the cluster to balance load.

Nodes can be connected together to form a cluster in many different topologies, we will discuss a couple of the more common topologies later in this chapter.

We'll also discuss client side load balancing, where we can balance client connections across the nodes of the cluster, and we'll consider message redistribution where HornetQ will redistribute messages between nodes to avoid starvation.

Another important part of clustering is *server discovery* where servers can broadcast their connection details so clients or other servers can connect to them with the minimum of configuration.

38.2. Server discovery

Server discovery is a mechanism by which servers can broadcast their connection settings across the network. This is useful for two purposes:

- Discovery by messaging clients. A messaging client wants to be able to connect to the servers of the cluster without having specific knowledge of which servers in the cluster are up at any one time. Messaging clients *can* be initialised with an explicit list of the servers in a cluster, but this is not flexible or maintainable as servers are added or removed from the cluster.
- Discovery by other servers. Servers in a cluster want to be able to create cluster connections to each other without having prior knowledge of all the other servers in the cluster.

Server discovery uses UDP [http://en.wikipedia.org/wiki/User_Datagram_Protocol] multicast to broadcast server connection settings. If UDP is disabled on your network you won't be able to use this, and will have to specify

servers explicitly when setting up a cluster or using a messaging client.

38.2.1. Broadcast Groups

A broadcast group is the means by which a server broadcasts connectors over the network. A connector defines a way in which a client (or other server) can make connections to the server. For more information on what a connector is, please see Chapter 16.

The broadcast group takes a set of connector pairs, each connector pair contains connection settings for a live and (optional) backup server and broadcasts them on the network. It also defines the UDP address and port settings.

Broadcast groups are defined in the server configuration file `hornetq-configuration.xml`. There can be many broadcast groups per HornetQ server. All broadcast groups must be defined in a `broadcast-groups` element.

Let's take a look at an example broadcast group from `hornetq-configuration.xml`:

```
<broadcast-groups>
   <broadcast-group name="my-broadcast-group">
      <local-bind-port>54321</local-bind-port>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <broadcast-period>1000</broadcast-period>
      <connector-ref connector-name="netty-connector"
        backup-connector-name="backup-connector"/>
   </broadcast-group>
</broadcast-groups>
```

Some of the broadcast group parameters are optional and you'll normally use the defaults, but we specify them all in the above example for clarity. Let's discuss each one in turn:

- `name` attribute. Each broadcast group in the server must have a unique name.
- `local-bind-address`. This is the local bind address that the datagram socket is bound to. If you have multiple network interfaces on your server, you would specify which one you wish to use for broadcasts by setting this property. If this property is not specified then the socket will be bound to the wildcard address, an IP address chosen by the kernel.
- `local-bind-port`. If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of `-1` which signifies that an anonymous port should be used.
- `group-address`. This is the multicast address to which the data will be broadcast. It is a class D IP address in the range `224.0.0.0` to `239.255.255.255`, inclusive. The address `224.0.0.0` is reserved and is not available for use. This parameter is mandatory.
- `group-port`. This is the UDP port number used for broadcasting. This parameter is mandatory.
- `broadcast-period`. This is the period in milliseconds between consecutive broadcasts. This parameter is optional, the default value is `1000` milliseconds.
- `connector-ref`. This specifies the connector and optional backup connector that will be broadcasted (see Chapter 16 for more information on connectors). The connector to be broadcasted is specified by the `connector-name` attribute, and the backup connector to be broadcasted is specified by the `backup-connector` attribute.

The `backup-connector` attribute is optional.

38.2.2. Discovery Groups

While the broadcast group defines how connector information is broadcasted from a server, a discovery group defines how connector information is received from a multicast address.

A discovery group maintains a list of connector pairs - one for each broadcast by a different server. As it receives broadcasts on the multicast group address from a particular server it updates its entry in the list for that server.

If it has not received a broadcast from a particular server for a length of time it will remove that server's entry from its list.

Discovery groups are used in two places in HornetQ:

- By cluster connections so they know what other servers in the cluster they should make connections to.
- By messaging clients so they can discovery what servers in the cluster they can connect to.

38.2.3. Defining Discovery Groups on the Server

For cluster connections, discovery groups are defined in the server side configuration file `hornetq-configuration.xml`. All discovery groups must be defined inside a `discovery-groups` element. There can be many discovery groups defined by HornetQ server. Let's look at an example:

```
<discovery-groups>
   <discovery-group name="my-discovery-group">
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

We'll consider each parameter of the discovery group:

- `name` attribute. Each discovery group must have a unique name per server.
- `group-address`. This is the multicast ip address of the group to listen on. It should match the `group-address` in the broadcast group that you wish to listen from. This parameter is mandatory.
- `group-port`. This is the UDP port of the multicast group. It should match the `group-port` in the broadcast group that you wish to listen from. This parameter is mandatory.
- `refresh-timeout`. This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the `broadcast-period` on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due to slight differences in timing. This parameter is optional, the default value is `10000` milliseconds (10 seconds).

38.2.4. Discovery Groups on the Client Side

Let's discuss how to configure a HornetQ client to use discovery to discover a list of servers to which it can connect. The way to do this differs depending on whether you're using JMS or the core API.

38.2.4.1. Configuring client discovery using JMS

If you're using JMS and you're also using the JMS Service on the server to load your JMS connection factory instances into JNDI, then you can specify which discovery group to use for your JMS connection factory in the server side xml configuration `hornetq-jms.xml`. Let's take a look at an example:

```
<connection-factory name="ConnectionFactory">
   <discovery-group-ref discovery-group-name="my-discovery-group"/>
    <entries>
       <entry name="ConnectionFactory"/>
    </entries>
</connection-factory>
```

The element `discovery-group-ref` specifies the name of a discovery group defined in `hornetq-configuration.xml`.

When this connection factory is downloaded from JNDI by a client application and JMS connections are created from it, those connections will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the discovery group configuration.

If you're using JMS, but you're not using JNDI to lookup a connection factory - you're instantiating the JMS connection factory directly then you can specify the discovery group parameters directly when creating the JMS connection factory. Here's an example:

```
final String groupAddress = "231.7.7.7";

final int groupPort = 9876;

ConnectionFactory jmsConnectionFactory =
        HornetQJMSClient.createConnectionFactory(groupAddress, groupPort);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();

Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

The `refresh-timeout` can be set directly on the connection factory by using the setter method `setDiscoveryRefreshTimeout() if you want to change the default value.`

There is also a further parameter settable on the connection factory using the setter method `setInitialWaitTimeout()`. If the connection factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is `2000` milliseconds.

38.2.4.2. Configuring client discovery using Core

If you're using the core API to directly instantiate `ClientSessionFactory` instances, then you can specify the discovery group parameters directly when creating the session factory. Here's an example:

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;
SessionFactory factory = HornetQClient.createClientSessionFactory(groupAddress, group
ClientSession session1 = factory.createClientSession(...); ClientSession
session2 = factory.createClientSession(...);
```

The `refresh-timeout` can be set directly on the session factory by using the setter method `setDiscoveryRefresh-Timeout() if you want to change the default value.`

There is also a further parameter settable on the session factory using the setter method `setInitialWaitTimeout()`. If the session factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the session factory will make sure it waits this long since creation before creating the first session. The default value for this parameter is `2000` milliseconds.

38.3. Server-Side Message Load Balancing

If cluster connections are defined between nodes of a cluster, then HornetQ will load balance messages arriving at a particular node from a client.

Let's take a simple example of a cluster of four nodes A, B, C, and D arranged in a *symmetric cluster* (described in Section 38.7.1). We have a queue called `OrderQueue` deployed on each node of the cluster.

We have client Ca connected to node A, sending orders to the server. We have also have order processor clients Pa, Pb, Pc, and Pd connected to each of the nodes A, B, C, D. If no cluster connection was defined on node A, then as order messages arrive on node A they will all end up in the `OrderQueue` on node A, so will only get consumed by the order processor client attached to node A, Pa.

If we define a cluster connection on node A, then as ordered messages arrive on node A instead of all of them going into the local `OrderQueue` instance, they are distributed in a round-robin fashion between all the nodes of the cluster. The messages are forwarded from the receiving node to other nodes of the cluster. This is all done on the server side, the client maintains a single connection to node A.

For example, messages arriving on node A might be distributed in the following order between the nodes: B, D, C, A, B, D, C, A, B, D. The exact order depends on the order the nodes started up, but the algorithm used is round robin.

HornetQ cluster connections can be configured to always blindly load balance messages in a round robin fashion irrespective of whether there are any matching consumers on other nodes, but they can be a bit cleverer than that and also be configured to only distribute to other nodes if they have matching consumers. We'll look at both these cases in turn with some examples, but first we'll discuss configuring cluster connections in general.

38.3.1. Configuring Cluster Connections

Cluster connections group servers into clusters so that messages can be load balanced between the nodes of the cluster. Let's take a look at a typical cluster connection. Cluster connections are always defined in `hornetq-configuration.xml` inside a `cluster-connection` element. There can be zero or more cluster connections defined

per HornetQ server.

```
<cluster-connections>
    <cluster-connection name="my-cluster">
        <address>jms</address>
        <retry-interval>500</retry-interval>
        <use-duplicate-detection>true</use-duplicate-detection>
        <forward-when-no-consumers>false</forward-when-no-consumers>
        <max-hops>1</max-hops>
        <discovery-group-ref discovery-group-name="my-discovery-group"/>
    </cluster-connection>
</cluster-connections>
```

In the above cluster connection all parameters have been explicitly specified. In practice you might use the defaults for some.

- `address`. Each cluster connection only applies to messages sent to an address that starts with this value.

  In this case, this cluster connection will load balance messages sent to address that start with `jms`. This cluster connection, will, in effect apply to all JMS queue and topic subscriptions since they map to core queues that start with the substring "jms".

  The address can be any value and you can have many cluster connections with different values of `address`, simultaneously balancing messages for those addresses, potentially to different clusters of servers. By having multiple cluster connections on different addresses a single HornetQ Server can effectively take part in multiple clusters simultaneously.

  By careful not to have multiple cluster connections with overlapping values of `address`, e.g. "europe" and "europe.news" since this could result in the same messages being distributed between more than one cluster connection, possibly resulting in duplicate deliveries.

  This parameter is mandatory.

- `retry-interval`. We mentioned before that, internally, cluster connections cause bridges to be created between the nodes of the cluster. If the cluster connection is created and the target node has not been started, or say, is being rebooted, then the cluster connections from other nodes will retry connecting to the target until it comes back up, in the same way as a bridge does.

  This parameter determines the interval in milliseconds between retry attempts. It has the same meaning as the `retry-interval` on a bridge (as described in Chapter 36).

  This parameter is optional and its default value is `500` milliseconds.

- `use-duplicate-detection`. Internally cluster connections use bridges to link the nodes, and bridges can be configured to add a duplicate id property in each message that is forwarded. If the target node of the bridge crashes and then recovers, messages might be resent from the source node. By enabling duplicate detection any duplicate messages will be filtered out and ignored on receipt at the target node.

  This parameter has the same meaning as `use-duplicate-detection` on a bridge. For more information on duplicate detection, please see Chapter 37.

  This parameter is optional and has a default value of `true`.

- `forward-when-no-consumers`. This parameter determines whether messages will be distributed round robin between other nodes of the cluster *irrespective* of whether there are matching or indeed any consumers on other nodes.

  If this is set to `true` then each incoming message will be round robin'd even though the same queues on the other nodes of the cluster may have no consumers at all, or they may have consumers that have non matching message filters (selectors). Note that HornetQ will *not* forward messages to other nodes if there are no *queues* of the same name on the other nodes, even if this parameter is set to `true`.

  If this is set to `false` then HornetQ will only forward messages to other nodes of the cluster if the address to which they are being forwarded has queues which have consumers, and if those consumers have message filters (selectors) at least one of those selectors must match the message.

  This parameter is optional, the default value is `false`.

- `max-hops`. When a cluster connection decides the set of nodes to which it might load balance a message, those nodes do not have to be directly connected to it via a cluster connection. HornetQ can be configured to also load balance messages to nodes which might be connected to it only indirectly with other HornetQ servers as intermediates in a chain.

  This allows HornetQ to be configured in more complex topologies and still provide message load balancing. We'll discuss this more later in this chapter.

  The default value for this parameter is `1`, which means messages are only load balanced to other HornetQ serves which are directly connected to this server. This parameter is optional.

- `discovery-group-ref`. This parameter determines which discovery group is used to obtain the list of other servers in the cluster that this cluster connection will make connections to.

38.3.2. Cluster User Credentials

When creating connections between nodes of a cluster to form a cluster connection, HornetQ uses a cluster user and cluster password which is defined in `hornetq-configuration.xml`:

```
<cluster-user>HORNETQ.CLUSTER.ADMIN.USER</cluster-user>
<cluster-password>CHANGE ME!!</cluster-password>
```

Warning

It is imperative that these values are changed from their default, or remote clients will be able to make connections to the server using the default values. If they are not changed from the default, HornetQ will detect this and pester you with a warning on every start-up.

38.4. Client-Side Load balancing

With HornetQ client-side load balancing, subsequent sessions created using a single session factory can be connected to different nodes of the cluster. This allows sessions to spread smoothly across the nodes of a cluster and not

be "clumped" on any particular node.

The load balancing policy to be used by the client factory is configurable. HornetQ provides two out-of-the-box load balancing policies and you can also implement your own and use that.

The out-of-the-box policies are

- Round Robin. With this policy the first node is chosen randomly then each subsequent node is chosen sequentially in the same order.

  For example nodes might be chosen in the order B, C, D, A, B, C, D, A, B or D, A, B, C, A, B, C, D, A or C, D, A, B, C, D, A, B, C, D, A.

- Random. With this policy each node is chosen randomly.

You can also implement your own policy by implementing the interface `org.hornetq.api.core.client.loadbalance.ConnectionLoadBalancingPolicy`

Specifying which load balancing policy to use differs whether you are using JMS or the core API. If you don't specify a policy then the default will be used which is `org.hornetq.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy`.

If you're using JMS, and you're using JNDI on the server to put your JMS connection factories into JNDI, then you can specify the load balancing policy directly in the `hornetq-jms.xml` configuration file on the server as follows:

```
<connection-factory name="ConnectionFactory">
    <discovery-group-ref discovery-group-name="my-discovery-group"/>
    <entries>
        <entry name="ConnectionFactory"/>
    </entries>
    <connection-load-balancing-policy-class-name>
    org.hornetq.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy
    </connection-load-balancing-policy-class-name>
</connection-factory>
```

The above example would deploy a JMS connection factory that uses the random connection load balancing policy.

If you're using JMS but you're instantiating your connection factory directly on the client side then you can set the load balancing policy using the setter on the `HornetQConnectionFactory` before using it:

```
ConnectionFactory jmsConnectionFactory = HornetQJMSClient.createConnectionFactory(...);
jmsConnectionFactory.setLoadBalancingPolicyClassName("com.acme.MyLoadBalancingPolicy");
```

If you're using the core API, you can set the load balancing policy directly on the `ClientSessionFactory` instance you are using:

```
ClientSessionFactory factory = HornetQClient.createClientSessionFactory(...);
factory.setLoadBalancingPolicyClassName("com.acme.MyLoadBalancingPolicy");
```

The set of servers over which the factory load balances can be determined in one of two ways:

- Specifying servers explicitly
- Using discovery.

38.5. Specifying Members of a Cluster Explicitly

Sometimes UDP is not enabled on a network so it's not possible to use UDP server discovery for clients to discover the list of servers in the cluster, or for servers to discover what other servers are in the cluster.

In this case, the list of servers in the cluster can be specified explicitly on each node and on the client side. Let's look at how we do this:

38.5.1. Specify List of Servers on the Client Side

This differs depending on whether you're using JMS or the Core API

38.5.1.1. Specifying List of Servers using JMS

If you're using JMS, and you're using the JMS Service to load your JMS connection factory instances directly into JNDI on the server, then you can specify the list of servers in the server side configuration file `hornetq-jms.xml`. Let's take a look at an example:

```
<connection-factory name="ConnectionFactory">
   <connectors>
      <connector-ref connector-name="my-connector1"
           backup-connector-name="my-backup-connector1"/>
      <connector-ref connector-name="my-connector2"
           backup-connector-name="my-backup-connector2"/>
      <connector-ref connector-name="my-connector3"
           backup-connector-name="my-backup-connector3"/>
   </connectors>
   <entries>
      <entry name="ConnectionFactory"/>
   </entries>
</connection-factory>
```

The `connection-factory` element can contain zero or more `connector-ref` elements, each one of which specifies a `connector-name` attribute and an optional `backup-connector-name` attribute. The `connector-name` attribute references a connector defined in `hornetq-configuration.xml` which will be used as a live connector. The `backup-connector-name` is optional, and if specified it also references a connector defined in `hornetq-configuration.xml`. For more information on connectors please see Chapter 16.

The connection factory thus maintains a list of [connector, backup connector] pairs, these pairs are then used by the client connection load balancing policy on the client side when creating connections to the cluster.

If you're using JMS but you're not using JNDI then you can also specify the list of [connector, backup connector] pairs directly when instantiating the `HornetQConnectionFactory`, here's an example:

```
List<Pair<TransportConfiguration, TransportConfiguration>> serverList =
```

```
        new ArrayList<Pair<TransportConfiguration, TransportConfiguration>>();

serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC0, backupTC0));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC1, backupTC1));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC2, backupTC2));

ConnectionFactory jmsConnectionFactory = HornetQJMSClient.createConnectionFactory(serverList);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();

Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

In the above snippet we create a list of pairs of `TransportConfiguration` objects. Each `TransportConfiguration` object contains knowledge of how to make a connection to a specific server.

A `HornetQConnectionFactory` instance is then created passing the list of servers in the constructor. Any connections subsequently created by this factory will create connections according to the client connection load balancing policy applied to that list of servers.

38.5.1.2. Specifying List of Servers using the Core API

If you're using the core API you can also specify the list of servers directly when creating the `ClientSessionFactory` instance. Here's an example:

```
List<Pair<TransportConfiguration, TransportConfiguration>> serverList =
        new ArrayList<Pair<TransportConfiguration, TransportConfiguration>>();

serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC0, backupTC0));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC1, backupTC1));
serverList.add(new Pair<TransportConfiguration,
        TransportConfiguration>(liveTC2, backupTC2));

ClientSessionFactory factory = HornetQClient.createClientSessionFactory(serverList);

ClientSession sesison1 = factory.createClientSession(...);

ClientSession session2 = factory.createClientSession(...);
```

In the above snippet we create a list of pairs of `TransportConfiguration` objects. Each `TransportConfiguration` object contains knowledge of how to make a connection to a specific server. For more information on this, please see Chapter 16.

A `ClientSessionFactoryImpl` instance is then created passing the list of servers in the constructor. Any sessions subsequently created by this factory will create sessions according to the client connection load balancing policy applied to that list of servers.

38.5.2. Specifying List of Servers to form a Cluster

Let's take a look at an example where each cluster connection is defined for a symmetric cluster, but we're not using discovery for each node to discover its neighbours, instead we'll configure each cluster connection to have explicit knowledge of all the other nodes in the cluster.

Here's an example cluster connection definition showing that:

```
<cluster-connections>
    <cluster-connection name="my-explicit-cluster">
        <address>jms</address>
        <connector-ref connector-name="my-connector1"
            backup-connector-name="my-backup-connector1"/>
        <connector-ref connector-name="my-connector2"
            backup-connector-name="my-backup-connector2"/>
        <connector-ref connector-name="my-connector3"
            backup-connector-name="my-backup-connector3"/>
    </cluster-connection>
</cluster-connections>
```

The `cluster-connection` element can contain zero or more `connector-ref` elements, each one of which specifies a `connector-name` attribute and an optional `backup-connector-name` attribute. The `connector-name` attribute references a connector defined in `hornetq-configuration.xml` which will be used as a live connector. The `backup-connector-name` is optional, and if specified it also references a connector defined in `hornetq-configuration.xml`. For more information on connectors please see Chapter 16.

Note

Due to a limitation in HornetQ 2.0.0, failover is not supported for clusters defined using a static set of nodes. To support failover over cluster nodes, they must be configured to use a discovery group.

38.6. Message Redistribution

Another important part of clustering is message redistribution. Earlier we learned how server side message load balancing round robins messages across the cluster. If `forward-when-no-consumers` is false, then messages won't be forwarded to nodes which don't have matching consumers, this is great and ensures that messages don't arrive on a queue which has no consumers to consume them, however there is a situation it doesn't solve: What happens if the consumers on a queue close after the messages have been sent to the node? If there are no consumers on the queue the message won't get consumed and we have a *starvation* situation.

This is where message redistribution comes in. With message redistribution HornetQ can be configured to automatically *redistribute* messages from queues which have no consumers back to other nodes in the cluster which do have matching consumers.

Message redistribution can be configured to kick in immediately after the last consumer on a queue is closed, or to wait a configurable delay after the last consumer on a queue is closed before redistributing. By default message redistribution is disabled.

Message redistribution can be configured on a per address basis, by specifying the redistribution delay in the address settings, for more information on configuring address settings, please see Chapter 25.

Here's an address settings snippet from `hornetq-configuration.xml` showing how message redistribution is enabled for a set of queues:

```
<address-settings>
   <address-setting match="jms.#">
      <redistribution-delay>0</redistribution-delay>
   </address-setting>
 </address-settings>
```

The above `address-settings` block would set a `redistribution-delay` of `0` for any queue which is bound to an address that starts with "jms.". All JMS queues and topic subscriptions are bound to addresses that start with "jms.", so the above would enable instant (no delay) redistribution for all JMS queues and topic subscriptions.

The attribute `match` can be an exact match or it can be a string that conforms to the HornetQ wildcard syntax (described in Chapter 13).

The element `redistribution-delay` defines the delay in milliseconds after the last consumer is closed on a queue before redistributing messages from that queue to other nodes of the cluster which do have matching consumers. A delay of zero means the messages will be immediately redistributed. A value of `-1` signifies that messages will never be redistributed. The default value is `-1`.

It often makes sense to introduce a delay before redistributing as it's a common case that a consumer closes but another one quickly is created on the same queue, in such a case you probably don't want to redistribute immediately since the new consumer will arrive shortly.

38.7. Cluster topologies

HornetQ clusters can be connected together in many different topologies, let's consider the two most common ones here

38.7.1. Symmetric cluster

A symmetric cluster is probably the most common cluster topology, and you'll be familiar with if you've had experience of JBoss Application Server clustering.

With a symmetric cluster every node in the cluster is connected to every other node in the cluster. In other words every node in the cluster is no more than one hop away from every other node.

To form a symmetric cluster every node in the cluster defines a cluster connection with the attribute `max-hops` set to `1`. Typically the cluster connection will use server discovery in order to know what other servers in the cluster it should connect to, although it is possible to explicitly define each target server too in the cluster connection if, for example, UDP is not available on your network.

With a symmetric cluster each node knows about all the queues that exist on all the other nodes and what consumers they have. With this knowledge it can determine how to load balance and redistribute messages around the nodes.

38.7.2. Chain cluster

With a chain cluster, each node in the cluster is not connected to every node in the cluster directly, instead the nodes form a chain with a node on each end of the chain and all other nodes just connecting to the previous and next nodes in the chain.

An example of this would be a three node chain consisting of nodes A, B and C. Node A is hosted in one network and has many producer clients connected to it sending order messages. Due to corporate policy, the order consumer clients need to be hosted in a different network, and that network is only accessible via a third network. In this setup node B acts as a mediator with no producers or consumers on it. Any messages arriving on node A will be

forwarded to node B, which will in turn forward them to node C where they can get consumed. Node A does not need to directly connect to C, but all the nodes can still act as a part of the cluster.

To set up a cluster in this way, node A would define a cluster connection that connects to node B, and node B would define a cluster connection that connects to node C. In this case we only want cluster connections in one direction since we're only moving messages from node A->B->C and never from C->B->A.

For this topology we would set `max-hops` to `2`. With a value of `2` the knowledge of what queues and consumers that exist on node C would be propagated from node C to node B to node A. Node A would then know to distribute messages to node B when they arrive, even though node B has no consumers itself, it would know that a further hop away is node C which does have consumers.

39

High Availability and Failover

We define high availability as the *ability for the system to continue functioning after failure of one or more of the servers*.

A part of high availability is *failover* which we define as the *ability for client connections to migrate from one server to another in event of server failure so client applications can continue to operate*.

39.1. Live - Backup Pairs

HornetQ allows pairs of servers to be linked together as *live - backup* pairs. In this release there is a single backup server for each live server. A backup server is owned by only one live server. Backup servers are not operational until failover occurs.

Before failover, only the live server is serving the HornetQ clients while the backup server remains passive. When clients fail over to the backup server, the backup server becomes active and starts to service the HornetQ clients.

39.1.1. HA modes

HornetQ provides two different modes for high availability, either by *replicating data* from the live server journal to the backup server or using a *shared store* for both servers.

Note

Only persistent message data will survive failover. Any non persistent message data will not be available after failover.

39.1.1.1. Data Replication

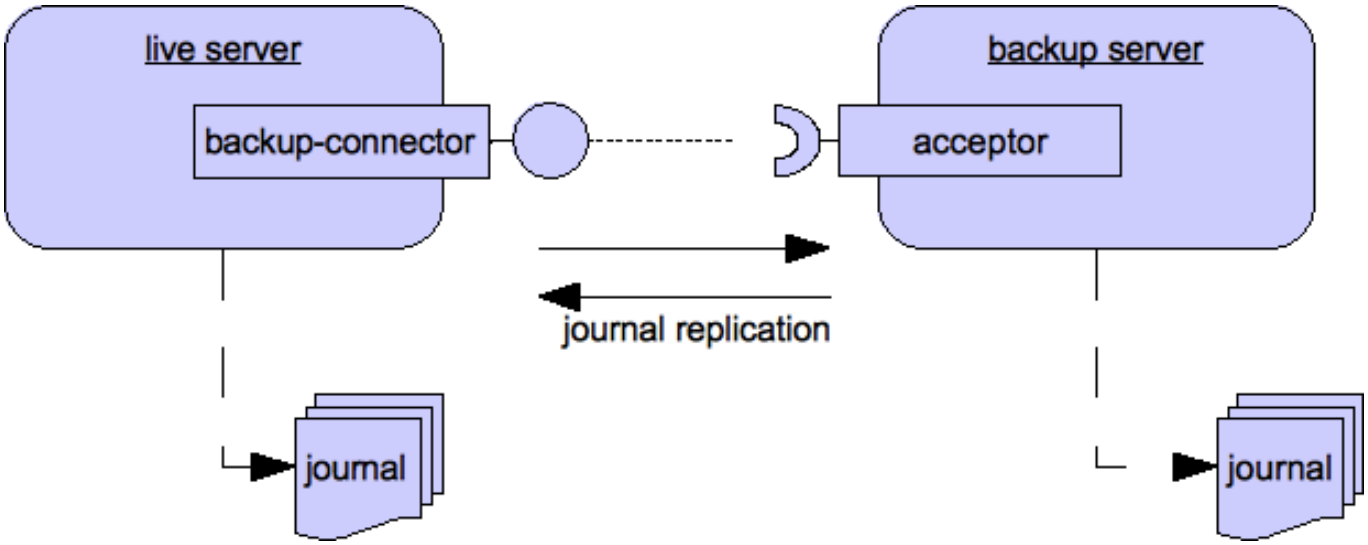
In this mode, data stored in the HornetQ journal are replicated from the live server's journal to the backup server's journal. Note that we do not replicate the entire server state, we only replicate the journal and other persistent operations.

Replication is performed in an asynchronous fashion between live and backup server. Data is replicated one way in a stream, and responses that the data has reached the backup is returned in another stream. Pipelining replications and responses to replications in separate streams allows replication throughput to be much higher than if we synchronously replicated data and waited for a response serially in an RPC manner before replicating the next piece of data.

When the user receives confirmation that a transaction has committed, prepared or rolled back or a durable message has been sent, we can guarantee it has reached the backup server and been persisted.

Data replication introduces some inevitable performance overhead compared to non replicated operation, but has the advantage in that it requires no expensive shared file system (e.g. a SAN) for failover, in other words it is a *shared-nothing* approach to high availability.

Failover with data replication is also faster than failover using shared storage, since the journal does not have to be reloaded on failover at the backup node.



39.1.1.1.1. Configuration

First, on the live server, in `hornetq-configuration.xml`, configure the live server with knowledge of its backup server. This is done by specifying a `backup-connector-ref` element. This element references a connector, also specified on the live server which specifies how to connect to the backup server.

Here's a snippet from live server's `hornetq-configuration.xml` configured to connect to its backup server:

```
<backup-connector-ref connector-name="backup-connector"/>

<connectors>
   <!-- This connector specifies how to connect to the backup server    -->
   <!-- backup server is located on host "192.168.0.11" and port "5445" -->
   <connector name="backup-connector">
     <factory-class>org.hornetq.integration.transports.netty.NettyConnectorFactory</factory-class>
     <param key="host" value="192.168.0.11"/>
     <param key="port" value="5445"/>
   </connector>
</connectors>
```

Secondly, on the backup server, we flag the server as a backup and make sure it has an acceptor that the live server can connect to. We also make sure the shared-store paramater is set to false:

```
<backup>true</backup>

<shared-store>false<shared-store>

<acceptors>
   <acceptor name="acceptor">
      <factory-class>org.hornetq.integration.transports.netty.NettyAcceptorFactory</factory-class>
      <param key="host" value="192.168.0.11"/>
      <param key="port" value="5445"/>
```

```
    </acceptor>
  </acceptors>
```

For a backup server to function correctly it's also important that it has the same set of bridges, predefined queues, cluster connections, broadcast groups and discovery groups as defined on the live node. The easiest way to ensure this is to copy the entire server side configuration from live to backup and just make the changes as specified above.

39.1.1.1.2. Synchronizing a Backup Node to a Live Node

In order for live - backup pairs to operate properly, they must be identical replicas. This means you cannot just use any backup server that's previously been used for other purposes as a backup server, since it will have different data in its persistent storage. If you try to do so, you will receive an exception in the logs and the server will fail to start.

To create a backup server for a live server that's already been used for other purposes, it's necessary to copy the `data` directory from the live server to the backup server. This means the backup server will have an identical persistent store to the backup server.

One a live server has failed over onto a backup server, the old live server becomes invalid and cannot just be restarted. To resynchonize the pair as a working live backup pair again, both servers need to be stopped, the data copied from the live node to the backup node and restarted again.

The next release of HornetQ will provide functionality for automatically synchronizing a new backup node to a live node without having to temporarily bring down the live node.

39.1.1.2. Shared Store

When using a shared store, both live and backup servers share the *same* journal using a shared file system.

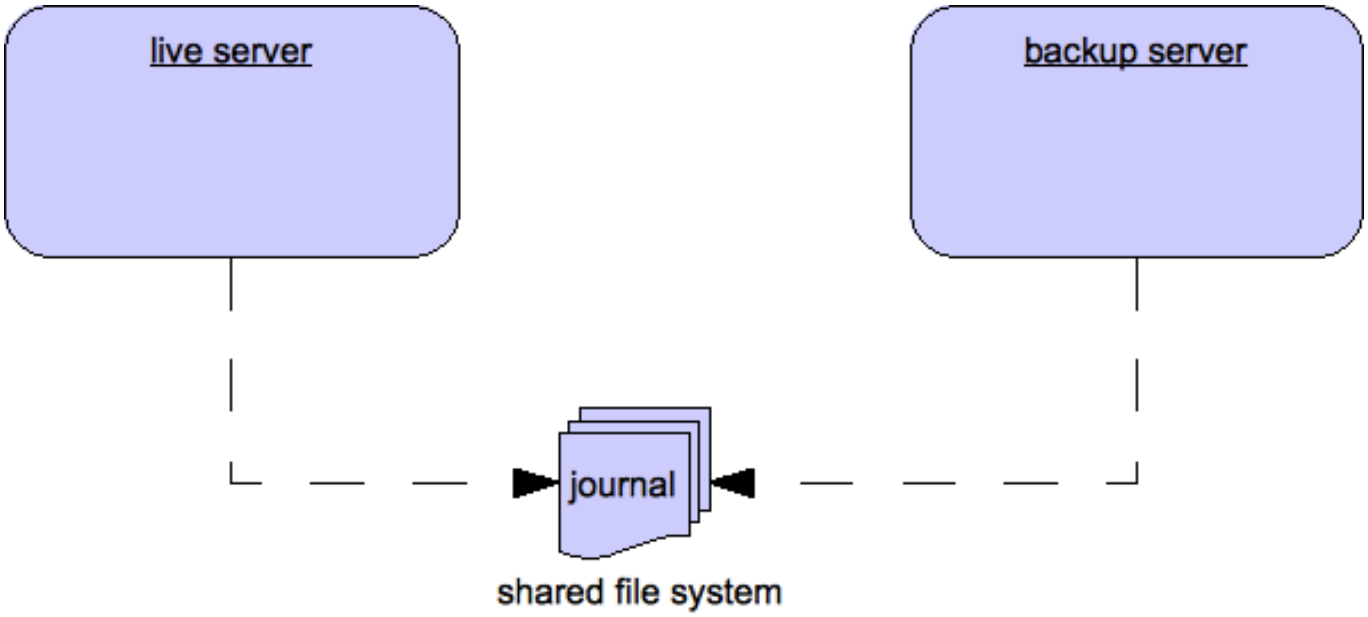
When failover occurs and the backup server takes over, it will load the persistent storage from the shared file system and clients can connect to it.

This style of high availability differs from data replication in that it requires a shared file system which is accessible by both the live and backup nodes. Typically this will be some kind of high performance Storage Area Network (SAN). We do not recommend you use Network Attached Storage (NAS), e.g. NFS mounts to store any shared journal (NFS is slow).

The advantage of shared-store high availability is that no replication occurs between the live and backup nodes, this means it does not suffer any performance penalties due to the overhead of replication during normal operation.

The disadvantage of shared store replication is that it requires a shared file system, and when the backup server activates it needs to load the journal from the shared store which can take some time depending on the amount of data in the store.

If you require the highest performance during normal operation, have access to a fast SAN, and can live with a slightly slower failover (depending on amount of data), we recommend shared store high availability



39.1.1.2.1. Configuration

To configure the live and backup server to share their store, configure both `hornetq-configuration.xml`:

```
<shared-store>true<shared-store>
```

In order for live - backup pairs to operate properly with a shared store, both servers must have configured the location of journal directory to point to the *same shared location* (as explained in Section 15.2)

If clients will use automatic failover with JMS, the live server will need to configure a connector to the backup server and reference it from its `hornetq-jms.xml` configuration as explained in Section 39.2.1.

39.1.1.2.2. Synchronizing a Backup Node to a Live Node

As both live and backup servers share the same journal, they do not need to be synchronized. However until, both live and backup servers are up and running, high-availability can not be provided with a single server. After failover, at first opportunity, stop the backup server (which is active) and restart the live and backup servers.

In the next release of HornetQ we will provide functionality to automatically synchronize a new backup server with a running live server without having to temporarily bring the live server down.

39.2. Failover Modes

HornetQ defines two types of client failover:

- Automatic client failover
- Application-level client failover

HornetQ also provides 100% transparent automatic reattachment of connections to the same server (e.g. in case of

transient network problems). This is similar to failover, except it's reconnecting to the same server and is discussed in Chapter 34

During failover, if the client has consumers on any non persistent or temporary queues, those queues will be automatically recreated during failover on the backup node, since the backup node will not have any knowledge of non persistent queues.

39.2.1. Automatic Client Failover

HornetQ clients can be configured with knowledge of live and backup servers, so that in event of connection failure at the client - live server connection, the client will detect this and reconnect to the backup server. The backup server will then automatically recreate any sessions and consumers that existed on each connection before failover, thus saving the user from having to hand-code manual reconnection logic.

HornetQ clients detect connection failure when it has not received packets from the server within the time given by `client-failure-check-period` as explained in section Chapter 17. If the client does not receive data in good time, it will assume the connection has failed and attempt failover.

HornetQ clients can be configured with the list of live-backup server pairs in a number of different ways. They can be configured explicitly or probably the most common way of doing this is to use *server discovery* for the client to automatically discover the list. For full details on how to configure server discovery, please see Section 38.2. Alternatively, the clients can explicitly specifies pairs of live-backup server as explained in Section 38.5.2.

To enable automatic client failover, the client must be configured to allow non-zero reconnection attempts (as explained in Chapter 34).

Sometimes you want a client to failover onto a backup server even if the live server is just cleanly shutdown rather than having crashed or the connection failed. To configure this you can set the property `FailoverOnServerShutdown` to true either on the `HornetQConnectionFactory` if you're using JMS or in the `hornetq-jms.xml` file when you define the connection factory, or if using core by setting the property directly on the `ClientSessionFactoryImpl` instance after creation. The default value for this property is `false`, this means that by default *HornetQ clients will not failover to a backup server if the live server is simply shutdown cleanly.*

Note

By default, cleanly shutting down the server **will not** trigger failover on the client.

Using CTRL-C on a HornetQ server or JBoss AS instance causes the server to **cleanly shut down**, so will not trigger failover on the client.

If you want the client to failover when its server is cleanly shutdown then you must set the property `FailoverOnServerShutdown` to true

For examples of automatic failover with transacted and non-transacted JMS sessions, please see Section 11.1.55 and Section 11.1.33.

39.2.1.1. A Note on Server Replication

HornetQ does not replicate full server state betwen live and backup servers. When the new session is automatically

recreated on the backup it won't have any knowledge of messages already sent or acknowledged in that session. Any in-flight sends or acknowledgements at the time of failover might also be lost.

By replicating full server state, theoretically we could provide a 100% transparent seamless failover, which would avoid any lost messages or acknowledgements, however this comes at a great cost: replicating the full server state (including the queues, session, etc.). This would require replication of the entire server state machine; every operation on the live server would have to replicated on the replica server(s) in the exact same global order to ensure a consistent replica state. This is extremely hard to do in a performant and scalable way, especially when one considers that multiple threads are changing the live server state concurrently.

It is possible to provide full state machine replication using techniques such as *virtual synchrony*, but this does not scale well and effectively serializes all operations to a single thread, dramatically reducing concurrency.

Other techniques for multi-threaded active replication exist such as replicating lock states or replicating thread scheduling but this is very hard to achieve at a Java level.

Consequently it xas decided it was not worth massively reducing performance and concurrency for the sake of 100% transparent failover. Even without 100% transparent failover, it is simple to guarantee *once and only once* delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions. However this is not 100% transparent to the client code.

39.2.1.2. Handling Blocking Calls During Failover

If the client code is in a blocking call to the server, waiting for a response to continue its execution, when failover occurs, the new session will not have any knowledge of the call that was in progress. This call might otherwise hang for ever, waiting for a response that will never come.

To prevent this, HornetQ will unblock any blocking calls that were in progress at the time of failover by making them throw a `javax.jms.JMSException` (if using JMS), or a `HornetQException` with error code `HornetQException.UNBLOCKED`. It is up to the client code to catch this exception and retry any operations if desired.

If the method being unblocked is a call to commit(), or prepare(), then the transaction will be automatically rolled back and HornetQ will throw a `javax.jms.TransactionRolledBackException` (if using JMS), or a `HornetQException` with error code `HornetQException.TRANSACTION_ROLLED_BACK` if using the core API.

39.2.1.3. Handling Failover With Transactions

If the session is transactional and messages have already been sent or acknowledged in the current transaction, then the server cannot be sure that messages sent or acknowledgements have not been lost during the failover.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit it will throw a `javax.jms.TransactionRolledBackException` (if using JMS), or a `HornetQException` with error code `HornetQException.TRANSACTION_ROLLED_BACK` if using the core API.

It is up to the user to catch the exception, and perform any client side local rollback code as necessary. The user can then just retry the transactional operations again on the same session.

HornetQ ships with a fully functioning example demonstrating how to do this, please see Section 11.1.55

If failover occurs when a commit call is being executed, the server, as previously described, will unblock the call to prevent a hang, since no response will come back. In this case it is not easy for the client to determine whether the

transaction commit was actually processed on the live server before failure occurred.

To remedy this, the client can simply enable duplicate detection (Chapter 37) in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction had indeed been committed on the live server successfully before failover, then when the transaction is retried, duplicate detection will ensure that any durable messages resent in the transaction will be ignored on the server to prevent them getting sent more than once.

Note

By catching the rollback exceptions and retrying, catching unblocked calls and enabling duplicate detection, once and only once delivery guarantees for messages can be provided in the case of failure, guaranteeing 100% no loss or duplication of messages.

39.2.1.4. Handling Failover With Non Transactional Sessions

If the session is non transactional, messages or acknowledgements can be lost in the event of failover.

If you wish to provide *once and only once* delivery guarantees for non transacted sessions too, enabled duplicate detection, and catch unblock exceptions as described in Section 39.2.1.2

39.2.2. Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: `java.jms.ExceptionListener`. Please consult the JMS javadoc or any good JMS tutorial for more information on how to use this.

The HornetQ core API also provides a similar feature in the form of the class `org.hornet.core.client.SessionFailureListener`

Any ExceptionListener or SessionFailureListener instance will always be called by HornetQ on event of connection failure, **irrespective** of whether the connection was successfully failed over, reconnected or reattached.

39.2.3. Application-Level Failover

In some cases you may not want automatic client failover, and prefer to handle any connection failure yourself, and code your own manually reconnection logic in your own failure handler. We define this as *application-level* failover, since the failover is handled at the user application level.

To implement application-level failover, if you're using JMS then you need to set an `ExceptionListener` class on the JMS connection. The `ExceptionListener` will be called by HornetQ in the event that connection failure is detected. In your `ExceptionListener`, you would close your old JMS connections, potentially look up new connection factory instances from JNDI and creating new connections. In this case you may well be using HA-JNDI [http://www.jboss.org/community/wiki/JBossHAJNDIImpl] to ensure that the new connection factory is looked up from a different server.

For a working example of application-level failover, please see Section 11.1.1.

If you are using the core API, then the procedure is very similar: you would set a `FailureListener` on the core `ClientSession` instances.

40

Libaio Native Libraries

HornetQ distributes a native library, used as a bridge between HornetQ and linux libaio.

`libaio` is a library, developed as part of the linux kernel project. With `libaio` we submit writes to the operating system where they are processed asynchronously. Some time later the OS will call our code back when they have been processed.

We use this in our high performance journal if configured to do so, please see Chapter 15.

These are the native libraries distributed by HornetQ:

- libHornetQAIO32.so - x86 32 bits
- libHornetQAIO64.so - x86 64 bits

When using libaio, HornetQ will always try loading these files as long as they are on the library path.

40.1. Compiling the native libraries

In the case that you are using Linux on a platform other than x86_32 or x86_64 (for example Itanium 64 bits or IBM Power) you may need to compile the native library, since we do not distribute binaries for those platforms with the release.

40.1.1. Install requirements

> **Note**
>
> At the moment the native layer is only available on Linux. If you are in a platform other than Linux the native compilation will not work

The native library uses autoconf [http://en.wikipedia.org/wiki/Autoconf] what makes the compilation process easy, however you need to install extra packages as a requirement for compilation:

- gcc - C Compiler
- gcc-c++ or g++ - Extension to gcc with support for C++
- autoconf - Tool for automating native build process
- make - Plain old make

- automake - Tool for automating make generation
- libtool - Tool for link editing native libraries
- libaio - library to disk asynchronous IO kernel functions
- libaio-dev - Compilation support for libaio
- A full JDK installed with the environment variable JAVA_HOME set to its location

To perform this installation on RHEL or Fedora, you can simply type this at a command line:

```
sudo yum install automake libtool autoconf gcc-g++ gcc libaio libaio-dev make
```

Or on debian systems:

```
sudo apt-get install automake libtool autoconf gcc-g++ gcc libaio libaio-dev make
```

Note

You could find a slight variation of the package names depending on the version and linux distribution. (for example gcc-c++ on Fedora versus g++ on Debian systems)

40.1.2. Invoking the compilation

In the distribution, in the `native-src` directory, execute the shell script `bootstrap`. This script will invoke `automake` and `make` what will create all the make files and the native library.

```
someUser@someBox:/messaging-distribution/native-src$ ./bootstrap
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p

...

configure: creating ./config.status
config.status: creating Makefile
config.status: creating ./src/Makefile
config.status: creating config.h
config.status: config.h is unchanged
config.status: executing depfiles commands
config.status: executing libtool commands

...
```

The produced library will be at `./native-src/src/.libs/libHornetQAIO.so`. Simply move that file over `bin` on the distribution or the place you have chosen on the library path.

If you want to perform changes on the HornetQ libaio code, you could just call make directly at the `native-src` directory.

41

Thread management

This chapter describes how HornetQ uses and pools threads and how you can manage them.

First we'll discuss how threads are managed and used on the server side, then we'll look at the client side.

41.1. Server-Side Thread Management

Each HornetQ Server maintains a single thread pool for general use, and a scheduled thread pool for scheduled use. A Java scheduled thread pool cannot be configured to use a standard thread pool, otherwise we could use a single thread pool for both scheduled and non scheduled activity.

There are also a small number of other places where threads are used directly, we'll discuss each in turn.

41.1.1. Server Scheduled Thread Pool

The server scheduled thread pool is used for most activities on the server side that require running periodically or with delays. It maps internally to a `java.util.concurrent.ScheduledThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `hornetq-configuration.xml` with the `scheduled-thread-pool-max-size` parameter. The default value is `5` threads. A small number of threads is usually sufficient for this pool.

41.1.2. General Purpose Server Thread Pool

This general purpose thread pool is used for most asynchronous actions on the server side. It maps internally to a `java.util.concurrent.ThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `hornetq-configuration.xml` with the `thread-pool-max-size` parameter.

If a value of `-1` is used this signifies that the thread pool has no upper bound and new threads will be created on demand if there are enough threads available to satisfy a request. If activity later subsides then threads are timed-out and closed.

If a value of `n` where `n`is a positive integer greater than zero is used this signifies that the thread pool is bounded. If more requests come in and there are no free threads in the pool and the pool is full then requests will block until a thread becomes available. It is recommended that a bounded thread pool is used with caution since it can lead to dead-lock situations if the upper bound is chosen to be too low.

The default value for `thread-pool-max-size` is `-1`, i.e. the thread pool is unbounded.

See the J2SE javadoc [http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html] for more information on unbounded (cached), and bounded (fixed) thread pools.

41.1.3. Expiry Reaper Thread

A single thread is also used on the server side to scan for expired messages in queues. We cannot use either of the thread pools for this since this thread needs to run at its own configurable priority.

For more information on configuring the reaper, please see Chapter 22.

41.1.4. Asynchronous IO

Asynchronous IO has a thread pool for receiving and dispatching events out of the native layer. You will find it on a thread dump with the prefix HornetQ-AIO-poller-pool. HornetQ uses one thread per opened file on the journal (there is usually one).

There is also a single thread used to invoke writes on libaio. We do that to avoid context switching on libaio what would cause performance issues. You will find this thread on a thread dump with the prefix HornetQ-AIO-writer-pool.

41.2. Client-Side Thread Management

On the client side, HornetQ maintains a single static scheduled thread pool and a single static general thread pool for use by all clients using the same classloader in that JVM instance.

The static scheduled thread pool has a maximum size of `5` threads, and the general purpose thread pool has an unbounded maximum size.

If required HornetQ can also be configured so that each `ClientSessionFactory` instance does not use these static pools but instead maintains its own scheduled and general purpose pool. Any sessions created from that `ClientSessionFactory` will use those pools instead.

To configure a `ClientSessionFactory` instance to use its own pools, simply use the appropriate setter methods immediately after creation, for example:

```
ClientSessionFactory myFactory = HornetQClient.createClientSessionFactory(...);
myFactory.setUseGlobalPools(false);
myFactory.setScheduledThreadPoolMaxSize(10);
myFactory.setThreadPoolMaxSize(-1);
```

If you're using the JMS API, you can set the same parameters on the ClientSessionFactory and use it to create the `ConnectionFactory` instance, for example:

```
ConnectionFactory myConnectionFactory = HornetQJMSClient.createConnectionFactory(myFactory);
```

If you're using JNDI to instantiate `HornetQConnectionFactory` instances, you can also set these parameters in the `hornetq-jms.xml` file where you describe your connection factory, for example:

```
<connection-factory name="ConnectionFactory">
```

```
    <connectors>
       <connector-ref connector-name="netty"/>
    </connectors>
    <entries>
        <entry name="ConnectionFactory"/>
        <entry name="XAConnectionFactory"/>
    </entries>
    <use-global-pools>false</use-global-pools>
    <scheduled-thread-pool-max-size>10</scheduled-thread-pool-max-size>
    <thread-pool-max-size>-1</thread-pool-max-size>
</connection-factory>
```

42

Logging

HornetQ has its own logging delegate that has no dependencies on any particular logging framework. The default delegate delegates all its logs to the standard JDK logging [http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/], (a.k.a Java-Util-Logging: JUL). By default the server picks up its JUL configuration from a `logging.properties` file found in the config directories. This is configured to use our own HornetQ logging formatter and will log to the console as well as a log file. For more information on configuring JUL visit Suns website.

You can configure a different Logging Delegate programatically or via a System Property.

To do this programatically simply do the following

```
org.hornetq.core.logging.Logger.setDelegateFactory(new Log4jLogDelegateFactory())
```

Where `Log4jLogDelegateFactory` is the implementation of `org.hornetq.spi.core.logging.LogDelegateFactory` that you would like to use.

To do this via a System Property simply set the property `org.hornetq.logger-delegate-factory-class-name` to the delegate factory being used, i.e.

```
-Dorg.hornetq.logger-delegate-factory-class-name=org.hornetq.integration.logging.Log4jLogDelegateFactory
```

As you can see in the above example HornetQ provides some Delegate Factories for your convenience. these are

1. org.hornetq.core.logging.impl.JULLogDelegateFactory - the default that uses JUL.
2. org.hornetq.integration.logging.Log4jLogDelegateFactory - which uses Log4J

If you configure your client's logging to use the JUL delegate, make sure you provide a `logging.properties` file and set the `java.util.logging.config.file` property on client startup

42.1. Logging With The JBoss Application Server

When HornetQ is deployed within the JBoss Application Server version 5.x or above then it will still use JUL however the logging is redirected to the default JBoss logger. For more information on this refer to the JBoss documentation. In versions before this you must specify what logger delegate you want to use.

43

Embedding HornetQ

HornetQ is designed as set of simple Plain Old Java Objects (POJOs). This means HornetQ can be instantiated and run in any dependency injection framework such as JBoss Microcontainer, Spring or Google Guice. It also means that if you have an application that could use messaging functionality internally, then it can *directly instantiate* HornetQ clients and servers in its own application code to perform that functionality. We call this *embedding* HornetQ.

Examples of applications that might want to do this include any application that needs very high performance, transactional, persistent messaging but doesn't want the hassle of writing it all from scratch.

Embedding HornetQ can be done in very few easy steps. Instantiate the configuration object, instantiate the server, start it, and you have a HornetQ running in your virtual machine. It's as simple and easy as that.

43.1. POJO instantiation

You can follow this step-by-step guide:

Create the configuration object - this contains configuration information for a HornetQ. If you want to configure it from a file on the classpath, use `FileConfigurationImpl`

```
import org.hornetq.core.config.Configuration;
import org.hornetq.core.config.impl.FileConfiguration;

...


Configuration config = new FileConfiguration();
config.setConfigurationUrl(urlToYourconfigfile);
config.start();
```

If you don't need to support a configuration file, just use `ConfigurationImpl` and change the config parameters accordingly, such as adding acceptors.

The acceptors are configured through `ConfigurationImpl`. Just add the `NettyAcceptorFactory` on the transports the same way you would through the main configuration file.

```
import org.hornetq.core.config.Configuration;
import org.hornetq.core.config.impl.ConfigurationImpl;

...

Configuration config = new ConfigurationImpl();
HashSet<TransportConfiguration> transports = new HashSet<TransportConfiguration>();

transports.add(new TransportConfiguration(NettyAcceptorFactory.class.getName()));
transports.add(new TransportConfiguration(InVMAcceptorFactory.class.getName()));
```

```
config.setAcceptorConfigurations(transports);
```

You need to instantiate and start HornetQ server. The class `org.hornetq.api.core.server.HornetQ` has a few static methods for creating servers with common configurations.

```
import org.hornetq.api.core.server.HornetQ;
import org.hornetq.core.server.HornetQServer;

...

HornetQServer server = HornetQ.newHornetQServer(config);

server.start();
```

You also have the option of instantiating `HornetQServerImpl` directly:

```
HornetQServer server =
        new HornetQServerImpl(config);
server.start();
```

43.2. Dependency Frameworks

You may also choose to use a dependency injection framework such as JBoss Micro Container™ or Spring Framework™.

HornetQ standalone uses JBoss Micro Container as the injection framework. `HornetQBootstrapServer` and `hornetq-beans.xml` which are part of the HornetQ distribution provide a very complete implementation of what's needed to bootstrap the server using JBoss Micro Container.

When using JBoss Micro Container, you need to provide an XML file declaring the `HornetQServer` and `Configuration` object, you can also inject a security manager and a MBean server if you want, but those are optional.

A very basic XML Bean declaration for the JBoss Micro Container would be:

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- The core configuration -->
   <bean name="Configuration"
         class="org.hornetq.core.config.impl.FileConfiguration">
   </bean>

           <!-- The core server -->
   <bean name="HornetQServer"
         class="org.hornetq.core.server.impl.HornetQServerImpl">
      <constructor>
         <parameter>
            <inject bean="Configuration"/>
         </parameter>
      </constructor>
   </bean>
   </deployment>
```

`HornetQBootstrapServer` provides an easy encapsulation of JBoss Micro Container.

```
HornetQBootstrapServer bootStrap =
        new HornetQBootstrapServer(new String[] {"hornetq-beans.xml"});
        bootStrap.run();
```

43.3. Connecting to the Embedded HornetQ

To connect clients to HornetQ you just create the factories as normal:

43.3.1. Core API

If using the core API, just create the `ClientSessionFactory` and use the regular core API.

```
ClientSessionFactory nettyFactory =  HornetQClient.createClientSessionFactory(
                                        new TransportConfiguration(
                                           InVMConnectorFactory.class.getName()));

ClientSession session = factory.createSession();

session.createQueue("example", "example", true);

ClientProducer producer = session.createProducer("example");

ClientMessage message = session.createMessage(true);

message.getBody().writeString("Hello");

producer.send(message);

session.start();

ClientConsumer consumer = session.createConsumer("example");

ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBody().readString());

session.close();
```

43.3.2. JMS API

Connection on an Embedded HornetQ through JMS is also simple. Just instantiate `ConnectionFactory` directly. The following example illustrates that.

```
ConnectionFactory cf =
    HornetQJMSClient.createConnectionFactory(
       new TransportConfiguration(InVMConnectorFactory.class.getName()));

Connection conn = cf.createConnection();

conn.start();

Session sess = conn.createSession(true, Session.SESSION_TRANSACTED);

MessageProducer prod = sess.createProducer(queue);

TextMessage msg = sess.createTextMessage("Hello!");
```

```
prod.send(msg);

sess.commit();

MessageConsumer consumer = sess.createConsumer(queue);

TextMessage txtmsg = (TextMessage)consumer.receive();

System.out.println("Msg = " + txtmsg.getText());

sess.commit();

conn.close();
```

43.4. JMS Embedding Example

Please see Section 11.2.1 for an example which shows how to setup and run HornetQ embedded with JMS.

44

Intercepting Operations

HornetQ supports *interceptors* to intercept packets entering the server. Any supplied interceptors would be called for any packet entering the server, this allows custom code to be executed, e.g. for auditing packets, filtering or other reasons. Interceptors can change the packets they intercept.

44.1. Implementing The Interceptors

A interceptor must implement the `Interceptor interface`:

```
package org.hornetq.api.core.interceptor;

public interface Interceptor
{
   boolean intercept(Packet packet, RemotingConnection connection)
                throws HornetQException;
}
```

The returned boolean value is important:

- if `true` is returned, the process continues normally
- if `false` is returned, the process is aborted, no other interceptors will be called and the packet will not be handled by the server at all.

44.2. Configuring The Interceptors

The interceptors are configured in `hornetq-configuration.xml`:

```
<remoting-interceptors>
   <class-name>org.hornetq.jms.example.LoginInterceptor</class-name>
   <class-name>org.hornetq.jms.example.AdditionalPropertyInterceptor</class-name>
</remoting-interceptors>
```

The interceptors classes (and their dependencies) must be added to the server classpath to be properly instantiated and called.

44.3. Interceptors on the Client Side

The interceptors can also be run on the client side to intercept packets *sent by the server* by adding the interceptor to the `ClientSessionFactory` with the `addInterceptor()` method.

The interceptors classes (and their dependencies) must be added to the client classpath to be properly instantiated and called from the client side.

44.4. Example

See Section 11.1.18 for an example which shows how to use interceptors to add properties to a message on the server.

45

Interoperability

45.1. Stomp and StompConnect

Stomp [http://stomp.codehaus.org/] is a wire protocol that allows Stomp clients to communicate with Stomp Brokers. StompConnect [http://stomp.codehaus.org/StompConnect] is a server that can act as a Stomp broker and proxy the Stomp protocol to the standard JMS API. Consequently, using StompConnect it is possible to turn HornetQ into a Stomp Broker and use any of the available stomp clients. These include clients written in C, C++, c# and .net etc.

To run StompConnect first start the HornetQ server and make sure that it is using JNDI.

Stomp requires the file `jndi.properties` to be available on the classpath. This should look something like:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Make sure this file is in the classpath along with the StompConnect jar and the HornetQ jars and simply run `java org.codehaus.stomp.jms.Main`.

HornetQ will shortly be implementing the Stomp protocol directly, so you won't have to use StompConnect to be able to use HornetQ with Stomp clients.

A list of STOMP clients is available here. [http://stomp.codehaus.org/Clients]

45.2. REST

REST support coming soon!

45.3. AMQP

AMQP support coming soon!

46

Performance Tuning

In this chapter we'll discuss how to tune HornetQ for optimum performance.

46.1. Tuning persistence

- Put the message journal on its own physical volume. If the disk is shared with other processes e.g. transaction co-ordinator, database or other journals which are also reading and writing from it, then this may greatly reduce performance since the disk head may be skipping all over the place between the different files. One of the advantages of an append only journal is that disk head movement is minimised - this advantage is destroyed if the disk is shared. If you're using paging or large messages make sure they're ideally put on separate volumes too.
- Minimum number of journal files. Set `journal-min-files` to a number of files that would fit your average sustainable rate. If you see new files being created on the journal data directory too often, i.e. lots of data is being persisted, you need to increase the minimal number of files, this way the journal would reuse more files instead of creating new data files.
- Journal file size. The journal file size should be aligned to the capacity of a cylinder on the disk. The default value 10MiB should be enough on most systems.
- Use AIO journal. If using Linux, try to keep your journal type as AIO. AIO will scale better than Java NIO.
- Tune `journal-buffer-timeout`. The timeout can be increased to increase throughput at the expense of latency.
- If you're running AIO you might be able to get some better performance by increasing `journal-max-io`. DO NOT change this parameter if you are running NIO.

46.2. Tuning JMS

There are a few areas where some tweaks can be done if you are using the JMS API

- Disable message id. Use the `setDisableMessageID()` method on the `MessageProducer` class to disable message ids if you don't need them. This decreases the size of the message and also avoids the overhead of creating a unique ID.
- Disable message timestamp. Use the `setDisableMessageTimeStamp()` method on the `MessageProducer` class to disable message timestamps if you don't need them.
- Avoid `ObjectMessage`. `ObjectMessage` is convenient but it comes at a cost. The body of a `ObjectMessage` uses Java serialization to serialize it to bytes. The Java serialized form of even small objects is very verbose so takes

up a lot of space on the wire, also Java serialization is slow compared to custom marshalling techniques. Only use `ObjectMessage` if you really can't use one of the other message types, i.e. if you really don't know the type of the payload until run-time.

- Avoid `AUTO_ACKNOWLEDGE`. `AUTO_ACKNOWLEDGE` mode requires an acknowledgement to be sent from the server for each message received on the client, this means more traffic on the network. If you can, use `DUPS_OK_ACKNOWLEDGE` or use `CLIENT_ACKNOWLEDGE` or a transacted session and batch up many acknowledgements with one acknowledge/commit.
- Avoid durable messages. By default JMS messages are durable. If you don't really need durable messages then set them to be non-durable. Durable messages incur a lot more overhead in persisting them to storage.

46.3. Other Tunings

There are various other places in HornetQ where we can perform some tuning:

- Use Asynchronous Send Acknowledgements. If you need to send durable messages non transactionally and you need a guarantee that they have reached the server by the time the call to send() returns, don't set durable messages to be sent blocking, instead use asynchronous send acknowledgements to get your acknowledgements of send back in a separate stream, see Chapter 20 for more information on this.
- Use pre-acknowledge mode. With pre-acknowledge mode, messages are acknowledged `before` they are sent to the client. This reduces the amount of acknowledgement traffic on the wire. For more information on this, see Chapter 29.
- Disable security. You may get a small performance boost by disabling security by setting the `security-enabled` parameter to `false` in `hornetq-configuration.xml`.
- Disable persistence. If you don't need message persistence, turn it off altogether by setting `persistence-enabled` to false in `hornetq-configuration.xml`.
- Sync transactions lazily. Setting `journal-sync-transactional` to `false` in `hornetq-configuration.xml` can give you better transactional persistent performance at the expense of some possibility of loss of transactions on failure. See Chapter 20 for more information.
- Sync non transactional lazily. Setting `journal-sync-non-transactional` to `false` in `hornetq-configuration.xml` can give you better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure. See Chapter 20 for more information.
- Send messages non blocking. Setting `block-on-durable-send` and `block-on-non-durable-send` to `false` in `hornetq-jms.xml` (if you're using JMS and JNDI) or directly on the ClientSessionFactory. This means you don't have to wait a whole network round trip for every message sent. See Chapter 20 for more information.
- Socket NIO vs Socket Old IO. By default HornetQ uses Socket NIO on the server and old (blocking) IO on the client side (see the chapter on configuring transports for more information Chapter 16). NIO is much more scalable but can give you some latency hit compared to old blocking IO. If you expect to be able to service many thousands of connections on the server, then continue to use NIO on the server. However, if don't expect many thousands of connections on the server you can configure the server acceptors to use old IO, and might get a small performance advantage.

- Use the core API not JMS. Using the JMS API you will have slightly lower performance than using the core API, since all JMS operations need to be translated into core operations before the server can handle them. If using the core API try to use methods that take `SimpleString` as much as possible. `SimpleString`, unlike java.lang.String does not require copying before it is written to the wire, so if you re-use `SimpleString` instances between calls then you can avoid some unnecessary copying.

46.4. Tuning Transport Settings

- Enable Nagle's algorithm [http://en.wikipedia.org/wiki/Nagle's_algorithm]. If you are sending many small messages, such that more than one can fit in a single IP packet thus providing better performance. This is done by setting `tcp-no-delay` to false with the Netty transports. See Chapter 16 for more information on this.

  Enabling Nagle's algorithm can make a very big difference in performance and is highly recommended if you're sending a lot of asynchronous traffice.

- TCP buffer sizes. If you have a fast network and fast machines you may get a performance boost by increasing the TCP send and receive buffer sizes. See the Chapter 16 for more information on this.

- Increase limit on file handles on the server. If you expect a lot of concurrent connections on your servers, or if clients are rapidly opening and closing connections, you should make sure the user running the server has permission to create sufficient file handles.

  This varies from operating system to operating system. On Linux systems you can increase the number of allowable open file handles in the file `/etc/security/limits.conf` e.g. add the lines

```
serveruser     soft    nofile  20000
serveruser     hard    nofile  20000
```

  This would allow up to 20000 file handles to be open by the user `serveruser`.

46.5. Tuning the VM

We highly recommend you use the latest Java 6 JVM, especially in the area of networking, many improvements have been made since Java 5. We test internally using the Sun JVM, so some of these tunings won't apply to JDKs from other providers (e.g. IBM or JRockit)

- Garbage collection. For smooth server operation we recommend using a parallel garbage collection algorithm, e.g. using the JVM argument `-XX:+UseParallelGC` on Sun JDKs.

- Memory settings. Give as much memory as you can to the server. HornetQ can run in low memory by using paging (described in Chapter 24) but if it can run with all queues in RAM this will improve performance. The amount of memory you require will depend on the size and number of your queues and the size and number of your messages. Use the JVM arguments `-Xms` and `-Xmx` to set server available RAM. We recommend setting them to the same high value.

  HornetQ will regularly sample JVM memory and reports if the available memory is below a configurable

threshold. Use this information to properly set JVM memory and paging. The sample is disabled by default. To enabled it, configure the sample frequency by setting `memory-measure-interval` in `hornetq-configuration.xml` (in milliseconds). When the available memory goes below the configured threshold, a warning is logged. The threshold can be also configured by setting `memory-warning-threshold` in `hornetq-configuration.xml` (default is 25%).

- Aggressive options. Different JVMs provide different sets of JVM tuning parameters, for the Sun Hotspot JVM the full list of options is available here [http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp]. We recommend at least using `-XX:+AggressiveOpts` and `-XX:+UseFastAccessorMethods`. You may get some mileage with the other tuning parameters depending on your OS platform and application usage patterns.

46.6. Avoiding Anti-Patterns

- Re-use connections / sessions / consumers / producers. Probably the most common messaging anti-pattern we see is users who create a new connection/session/producer for every message they send or every message they consume. This is a poor use of resources. These objects take time to create and may involve several network round trips. Always re-use them.

  Note

  Some popular libraries such as the Spring JMS Template are known to use these anti-patterns. If you're using Spring JMS Template and you're getting poor performance you know why. Don't blame HornetQ!

- Avoid fat messages. Verbose formats such as XML take up a lot of space on the wire and performance will suffer as result. Avoid XML in message bodies if you can.

- Don't create temporary queues for each request. This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern a message is sent to a target and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message they process it then send back a response to the address specified in the reply-to. A common mistake made with this pattern is to create a new temporary queue on each message sent. This will drastically reduce performance. Instead the temporary queue should be re-used for many requests.

- Don't use Message-Driven Beans for the sake of it. As soon as you start using MDBs you are greatly increasing the codepath for each message received compared to a straightforward message consumer, since a lot of extra application server code is executed. Ask yourself do you really need MDBs? Can you accomplish the same task using just a normal message consumer?

47

Configuration Reference

This section is a quick index for looking up configuration. Click on the element name to go to the specific chapter.

47.1. Server Configuration

47.1.1. hornetq-configuration.xml

This is the main core server configuration file.

Table 47.1. Server Configuration

Element Name	Element Type	Description	Default
backup	Boolean	true means that this server is a backup to another node in the cluster	false
backup-connector-ref	String	the name of the remoting connector to connect to the backup node	
bindings-directory	String	the directory to store the persisted bindings to	data/bindings
clustered	Boolean	true means that the server is clustered	false
connection-ttl-override	Long	if set, this will override how long (in ms) to keep a connection alive without receiving a ping.	-1
create-bindings-dir	Boolean	true means that the server will create the bindings directory on start up	true
create-journal-dir	Boolean	true means that the journal directory will be created	true
file-deployment-enabled	Boolean	true means that the server will load configuration	true

Element Name	Element Type	Description	Default
		from the configuration files	
id-cache-size	Integer	the size of the cache for pre creating message id's	2000
journal-buffer-size	Long	The size of the internal buffer on the journal.	128 KiB
journal-buffer-timeout	Long	The timeout (in nano-seconds) used to flush internal buffers on the journal.	20000
journal-compact-min-files	Integer	The minimal number of data files before we can start compacting	10
journal-compact-percentage	Integer	The percentage of live data on which we consider compacting the journal	30
journal-directory	String	the directory to store the journal files in	data/journal
journal-file-size	Long	the size (in bytes) of each journal file	128 * 1024
journal-max-io	Integer	the maximum number of write requests that can be in the AIO queue at any one time	500
journal-min-files	Integer	how many journal files to pre-create	2
journal-sync-transactional	Boolean	if true wait for transaction data to be synchronized to the journal before returning response to client	true
journal-sync-non-transactional	Boolean	if true wait for non transaction data to be synced to the journal before returning response to client.	true
journal-type	ASYNCIO|NIO	the type of journal to use	ASYNCIO
jmx-management-enabled	Boolean	true means that the management API is available via JMX	true

Element Name	Element Type	Description	Default
jmx-domain	String	the JMX domain used to registered HornetQ MBeans in the MBeanServer	org.hornetq
large-messages-directory	String	the directory to store large messages	data/largemessages
management-address	String	the name of the management address to send management messages to	jms.queue.hornetq.management
cluster-user	String	the user used to for replicating management operations between clustered nodes	HORNETQ.CLUSTER.ADMIN.USER
cluster-password	String	the password used to for replicating management operations between clustered nodes	CHANGE ME!!
management-notification-address	String	the name of the address that consumers bind to receive management notifications	hornetq.notifications
message-counter-enabled	Boolean	true means that message counters are enabled	false
message-counter-max-day-history	Integer	how many days to keep message counter history	10
message-counter-sample-period	Long	the sample period (in ms) to use for message counters	10000
message-expiry-scan-period	Long	how often (in ms) to scan for expired messages	30000
message-expiry-thread-priority	Integer	the priority of the thread expiring messages	3
paging-directory	String	the directory to store paged messages in	data/paging
persist-delivery-count-before-delivery	Boolean	true means that the delivery count is persisted before delivery. False means that this only happens after a message has been cancelled.	false

Element Name	Element Type	Description	Default
persistence-enabled	Boolean	true means that the server will use the file based journal for persistence.	true
persist-id-cache	Boolean	true means that id's are persisted to the journal	true
scheduled-thread-pool-max-size	Integer	the number of threads that the main scheduled thread pool has.	5
security-enabled	Boolean	true means that security is enabled	true
security-invalidation-interval	Long	how long (in ms) to wait before invalidating the security cache	10000
thread-pool-max-size	Integer	the number of threads that the main thread pool has. -1 means no limit	-1
async-connection-execution-enabled	Boolean	Should incoming packets on the server be handed off to a thread from the thread pool for processing or should they be handled on the remoting thread?	true
transaction-timeout	Long	how long (in ms) before a transaction can be removed from the resource manager after create time	60000
transaction-timeout-scan-period	Long	how often (in ms) to scan for timeout transactions	1000
wild-card-routing-enabled	Boolean	true means that the server supports wild card routing	true
memory-measure-interval	Long	frequency to sample JVM memory in ms (or -1 to disable memory sampling)	-1
memory-warning-threshold	Integer	Percentage of available memory which threshold a warning log	25
acceptors	Acceptor	a list of remoting acceptors to create	

Element Name	Element Type	Description	Default
broadcast-groups	BroadcastGroup	a list of broadcast groups to create	
connectors	Connector	a list of remoting connectors configurations to create	
discovery-groups	DiscoveryGroup	a list of discovery groups to create	
diverts	Divert	a list of diverts to use	
divert.name (attribute)	String	a unique name for the divert - mandatory	
divert.routing-name	String	the routing name for the divert - mandatory	
divert.address	String	the address this divert will divert from - mandatory	
divert.forwarding-address	String	the forwarding address for the divert - mandatory	
divert.exclusive	Boolean	is this divert exclusive?	false
divert.filter	String	an optional core filter expression	null
divert.transformer-class-name	String	an optional class name of a transformer	
queues	Queue	a list of pre configured queues to create	
queues.name (attribute)	String	unique name of this queue	
queues.address	String	address for this queue - mandatory	
queues.filter	String	optional core filter expression for this queue	null
queues.durable	Boolean	is this queue durable?	true
bridges	Bridge	a list of bridges to create	
bridges.name (attribute)	String	unique name for this bridge	
bridges.queue-name	String	name of queue that this bridge consumes from -	

Element Name	Element Type	Description	Default
		mandatory	
bridges.forwarding-address	String	address to forward to. If omitted original address is used	null
bridges.filter	String	optional core filter expression	null
bridges.transformer-class-name	String	optional name of transformer class	null
bridges.retry-interval	Long	period (in ms) between successive retries	2000 ms
bridges.retry-interval-multiplier	Double	multiplier to apply to successive retry intervals	1.0
bridges.reconnect-attempts	Integer	maximum number of retry attempts, -1 signifies infinite	-1
bridges.failover-on-server-shutdown	Boolean	should failover be prompted if target server is cleanly shutdown?	false
bridges.use-duplicate-detection	Boolean	should duplicate detection headers be inserted in forwarded messages?	true
bridges.discovery-group-ref	String	name of discovery group used by this bridge	null
bridges.connector-ref.connector-name (attribute)	String	name of connector to use for live connection	
bridges.connector-ref.backup-connector-name (attribute)	String	optional name of connector to use for backup connection	null
cluster-connections	ClusterConnection	a list of cluster connections	
cluster-connections.name (attribute)	String	unique name for this cluster connection	
cluster-connections.address	String	name of address this cluster connection applies to	
cluster-connections.forward-when-no-consumers	Boolean	should messages be load balanced if there are no matching consumers on target?	false

Element Name	Element Type	Description	Default
cluster-connections.max-hops	Integer	maximum number of hops cluster topology is propagated	1
cluster-connections.retry-interval	Long	period (in ms) between successive retries	2000
cluster-connections.use-duplicate-detection	Boolean	should duplicate detection headers be inserted in forwarded messages?	true
cluster-connections.discovery-group-ref	String	name of discovery group used by this bridge	null
cluster-connections.connector-ref.connector-name (attribute)	String	name of connector to use for live connection	
cluster-connections.connector-ref.backup-connector-name (attribute)	String	optional name of connector to use for backup connection	null
security-settings	SecuritySetting	a list of security settings	
security-settings.match (attribute)	String	the string to use for matching security against an address	
security-settings.permission	Security Permission	a permision to add to the address	
security-settings.permission.type (attribute)	Permission Type	the type of permission	
security-settings.permission.roles (attribute)	Roles	a comma-separated list of roles to apply the permission to	
address-settings	AddressSetting	a list of address settings	
address-settings.dead-letter-address	String	the address to send dead messages too	
address-settings.max-delivery-attempts	Integer	how many times to attempt to deliver a message before sending to dead letter address	10
address-settings.expiry-address	String	the address to send expired messages too	
address-set-	Long	the time (in ms) to wait	0

Element Name	Element Type	Description	Default
tings.redelivery-delay		before redelivering a cancelled message.	
address-settings.last-value-queue	boolean	whether to treat the queue as a last value queue	false
address-settings.page-size-bytes	Long	the page size (in bytes) to use for an address	10 * 1024 * 1024
address-settings.max-size-bytes	Long	the maximum size (in bytes) to use in paging for an address	-1
address-settings.redistribution-delay	Long	how long (in ms) to wait after the last consumer is closed on a queue before redistributing messages.	-1

47.1.2. hornetq-jms.xml

This is the configuration file used by the server side JMS service to load JMS Queues, Topics and Connection Factories.

Table 47.2. JMS Server Configuration

Element Name	Element Type	Description	Default
connection-factory	ConnectionFactory	a list of connection factories to create and add to JNDI	
connection-factory.auto-group	Boolean	whether or not message grouping is automatically used	false
connection-factory.connectors	String	A list of connectors used bu the connection factory	
connection-factory.connectors.connector-ref.connector-name (attribute)	String	Name of the connector to connect to the live server	
connection-factory.connectors.connector-ref.backup-connector-name (attribute)	String	Name of the connector to connect to the backup server	
connection-factory.discovery-group-ref.d	String	Name of discovery group used by this connection	

Element Name	Element Type	Description	Default
iscovery-group-name (attribute)		factory	
connection-factory.discovery-initial-wait-timeout	Long	the initial time to wait (in ms) for discovery groups to wait for broadcasts	2000
connection-factory.block-on-acknowledge	Boolean	whether or not messages are acknowledged synchronously	false
connection-factory.block-on-non-durable-send	Boolean	whether or not non-durable messages are sent synchronously	false
connection-factory.block-on-durable-send	Boolean	whether or not durable messages are sent synchronously	true
connection-factory.call-timeout	Long	the timeout (in ms) for remote calls	30000
connection-factory.client-failure-check-period	Long	the period (in ms) after which the client will consider the connection failed after not receiving packets from the server	5000
connection-factory.client-id	String	the pre-configured client ID for the connection factory	null
connection-factory.connection-load-balancing-policy-class-name	String	the name of the load balancing class	org.hornetq.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy
connection-factory.connection-ttl	Long	the time to live (in ms) for connections	1 * 60000
connection-factory.consumer-max-rate	Integer	the fastest rate a consumer may consume messages per second	-1
connection-factory.consumer-window-size	Integer	the window size (in bytes) for consumer flow control	1024 * 1024
connection-factory.dups-ok-batch-size	Integer	the batch size (in bytes) between acknowledgements when using DUPS_OK_ACKNOWLEDGE mode	1024 * 1024

Element Name	Element Type	Description	Default
connection-factory.failover-on-server-shutdown	Boolean	whether or not to failover on server shutdown	false
connection-factory.min-large-message-size	Integer	the size (in bytes) before a message is treated as large	100 * 1024
connection-factory.cache-large-message-client	Boolean	If true clients using this connection factory will hold the large message body on temporary files.	false
connection-factory.pre-acknowledge	Boolean	whether messages are pre acknowledged by the server before sending	false
connection-factory.producer-max-rate	Integer	the maximum rate of messages per second that can be sent	-1
connection-factory.producer-window-size	Integer	the window size in bytes for producers sending messages	1024 * 1024
connection-factory.confirmation-window-size	Integer	the window size (in bytes) for reattachment confirmations	1024 * 1024
connection-factory.reconnect-attempts	Integer	maximum number of retry attempts, -1 signifies infinite	0
connection-factory.retry-interval	Long	the time (in ms) to retry a connection after failing	2000
connection-factory.retry-interval-multiplier	Double	multiplier to apply to successive retry intervals	1d
connection-factory.max-retry-interval	Integer	The maximum retry interval in the case a retry-interval-multiplier has been specified	2000
connection-factory.scheduled-thread-pool-max-size	Integer	the size of the scheduled thread pool	5
connection-factory.thread-pool-max-size	Integer	the size of the thread pool	-1
connection-fact-	Integer	the batch size (in bytes)	1024 * 1024

Element Name	Element Type	Description	Default
ory.transaction-batch-size		between acknowledgements when using a transactional session	
connection-factory.use-global-pools	Boolean	whether or not to use a global thread pool for threads	true
queue	Queue	a queue to create and add to JNDI	
queue.name (attribute)	String	unique name of the queue	
queue.entry	String	context where the queue will be bound in JNDI (there can be many)	
queue.durable	Boolean	is the queue durable?	true
queue.filter	String	optional filter expression for the queue	
topic	Topic	a topic to create and add to JNDI	
topic.name (attribute)	String	unique name of the topic	
topic.entry	String	context where the topic will be bound in JNDI (there can be many)	