JBoss Enterprise Application Platform 5

HornetQ User Guide

for use with JBoss Enterprise Application Platform 5

Edition 5.2.0

Last Updated: 2017-10-13

JBoss Enterprise Application Platform 5 HornetQ User Guide

for use with JBoss Enterprise Application Platform 5
Edition 5.2.0

Andy Taylor

Jared Morgan

Laura Bailey

Rebecca Newton

Edited by

Eva Kopalova

Petr Penicka

Russell Dickenson

Scott Mumford

Legal Notice

Abstract

A guide to HornetQ for JBoss Enterprise Application Platform 5 and its patch releases.

Table of Contents

CHAPTER 1. INTRODUCTION

JBoss Enterprise Application Platform messaging is based on HornetQ, which is an Open Source Message Oriented Middleware (MoM) project developed by the JBoss Community.

The Red Hat certified HornetQ product differs from the community offering because it is tested to ensure reliability on a number of hardware platforms, and is backed by a dedicated support network for product updates and technical support.

HornetQ is a Java-based multi-protocol, clustered, asynchronous messaging system. It features a high-performance journaling system suitable for persistent messaging implementations.

The High Availability (HA) functionality offered by HornetQ supports client fail-over in the event of server node failure.

CHAPTER 2. MIGRATING TO HORNETQ

Read this chapter to migrate existing JBoss Messaging applications to HornetQ. The *Installation Guide*contains instructions on installing an instance of JBoss Enterprise Application Platform with HornetQ, and should be read in conjunction with this section.

2.1. BEFORE YOU MIGRATE

IMPORTANT

You must shut down your client and server before attempting to migrate from JBoss Messaging to HornetQ.

NOTE

JBoss Messaging uses a database to store persistent data unless null persistence is specified. HornetQ uses its own high-performance journal system instead of a database, so your database does not need to be shut down for migration purposes.

WARNING

Due to the way NFS's synchronous locking mechanism works it is not an appropriate method of storing JMS data for use with HornetQ.

2.1.1. Back up relevant data

It is important to back up all data used in your application and your JBoss Messaging server before you migrate. This section outlines the data recommended for backup before migration.

2.1.1.1. JBoss Messaging database tables

JBoss Messaging uses a number of database tables to store persistent data. These tables include internal state information of JBoss Messaging, persistent messages and security settings. This section lists tables that hold important data.

JBM_MSG_REF, JBM_MSG

These tables store persistent messages and their states.

JBM_TX, JBM_TX_EX

These tables store transaction states.

JBM_USER, JBM_ROLE

These tables store user and role information.

JBM_POSTOFFICE

This table holds bindings information.

2.1.1.2. JBoss Messaging configuration files

Most configuration files are stored in **`$JBOSS_HOME/server/$PROFILE/deploy/messaging`**, assuming your JBoss Messaging server profile is **`messaging`**. Applications can choose other locations in which to deploy some configuration files.

You will need to back up and migrate the following configuration files to HornetQ:

Connection Factory service configuration files

Contain JMS connection factories deployed with the JBoss Messaging server.

Destination service configuration files

Contain JMS queues and topics deployed with the JBoss Messaging server.

Bridge service configuration files

Contain bridge services deployed with the JBoss Messaging server.

Other configuration files, such as **`messaging-service.xml`** and the database persistence configuration file, are JBoss Messaging MBeans configurations. The HornetQ implementation consists only of Plain Old Java Objects (POJOs), so these configuration files are not migration targets.

JBoss Messaging relies on the JBoss Remoting and JGroups services in order to work. The configuration files for these services contain settings specific to applications. HornetQ's transport layer and cluster design differ from that in JBoss Messaging. You will need to map the parameters in the service configuration files to their HornetQ equivalents, if any.

2.2. APPLICATION CODE

If you are using standard JMS in your application, you will need to make modifications to your source code regarding High Availability (HA) and clustering. Specifically you will need to adjust how your application handles failures and rollbacks. For more information, refer to Section 2.3, "Client-side Failure Handling".

If you are using JBoss Messaging proprietary features, such as ordering groups, you will need to adapt them to HornetQ equivalent features. Table 2.1, "Implementation class mapping between JBoss Messaging and HornetQ" lists the JBoss Messaging JMS implementation classes and their corresponding HornetQ equivalents.

Table 2.1. Implementation class mapping between JBoss Messaging and HornetQ

org.jboss.jms.client. Classname	HornetQ Equivalent Classname
`JBossConnectionFactory`	`org.hornetq.jms.client.HornetQConnectionFactory`
`JBossConnection`	`org.hornetq.jms.client.HornetQConnection`
`JBossSession`	`org.hornetq.jms.client.HornetQSession`
`JBossMessageProducer`	`org.hornetq.jms.client.HornetQMessageProducer`
`JBossMessageConsumer`	`org.hornetq.jms.client.HornetQMessageConsumer`

NOTE

Unless you have used JBoss Messaging-specific APIs, it is not necessary to explicitly case your JMS objects to specific implementations. You can just use the standard JMS APIs wherever possible.

2.3. CLIENT-SIDE FAILURE HANDLING

The JMS specification does not control how the server should behave on High Availability fail over, or clustering. However, the specification does recommend how an application must handle failures and rollback scenarios.

JBoss Messaging does not throw an exception to the client during fail over. Applications in JBoss Messaging are automatically connected to another node in the cluster.

In HornetQ, fail over or rollback triggers an exception. Your JBoss Messaging application must capture this exception and retry sending the message for it to be compatible with HornetQ.

In JBoss Messaging, the following expression was sufficient to handle failures:

```
producer.send(createMessage(session, i));
System.out.println("Message: " + i);
```

In HornetQ, the following try block is required:

```
try {
  producer.send(createMessage(session, i));
  System.out.println("Message: " + i);
} catch (Exception e) {
  Thread.sleep(1000);
  producer.send(createMessage(session, i));
}
```

2.4. INSTALLING HORNETQ

Refer to the ***JBoss Enterprise Application Platform*** *Installation Guide* available at https://access.redhat.com/knowledge/docs/ for up-to-date details on installing HornetQ with this distribution of JBoss Enterprise Application Platform.

2.5. SERVER CONFIGURATION MIGRATION

HornetQ configuration differs significantly from JBoss Messaging, so it is not possible to provide a one-to-one mapping between the two. Table 2.2, "Server attribute mapping between JBoss Messaging and HornetQ" lists JBoss Messaging server attributes and their equivalents (where possible) in HornetQ.

Unless explicitly indicated, attributes in the HornetQ Server Attributes column are configured in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**. The full set of supported directives in this file are documented in Appendix A, *Configuration Reference*

Table 2.2. Server attribute mapping between JBoss Messaging and HornetQ

JBoss Messaging Server Attributes (Server Peer MBean)	Equivalent HornetQ Server Attributes
`ServerPeerID`	N/A - HornetQ does not require a specified server ID
`DefaultQueueJNDIContext`, `DefaultTopicJNDIContext`	N/A
`PostOffice`	N/A
`DefaultDLQ`	N/A - HornetQ defines dead letter addresses at the core level. There is no default dead letter address for an address unless you specify one.
`DefaultMaxDeliveryAttempts`	N/A - In HornetQ, the default is always `10`.
`DefaultExpiryQueue`	N/A - HornetQ defines expiry addresses at the core level. There is no default expiry address for an address unless you specify one.
`DefaultRedeliveryDelay`	N/A - HornetQ's default redelivery delay is always `0` (no delay).
`MessageCounterSamplePeriod`	`message-counter-sample-period`
`FailoverStartTimeout`	N/A
`FailoverCompleteTimeout`	N/A
`DefaultMessageCounterHistoryDayLimit`	N/A
`ClusterPullConnectionFactory`	N/A
`DefaultPreserveOrdering`	N/A
`RecoverDeliveriesTimeout`	N/A
`EnableMessageCounters`	`message-counter-enabled`
`SuckerPassword`	`cluster-password`
`SuckerConnectionRetryTimes`	`bridges.reconnect-attempts`
`SuckerConnectionRetryInterval`	bridges.reconnect-interval
`StrictTCK`	N/A

JBoss Messaging Server Attributes (Server Peer MBean)	Equivalent HornetQ Server Attributes
`Destinations`, **`MessageCounters`**, **`MessageStatistics`**	N/A - These are part of HornetQ's management functions. Refer to the appropriate chapter for details.
`SupportsFailover`	N/A
`PersistenceManager`	N/A - HornetQ uses its built-in high-performance journal as its persistence utility.
`JMSUserManager`	N/A
`SecurityStore`	N/A - The security manager is configured in **`hornetq-beans.xml`** or **`hornetq-jboss-beans.xml`**.

2.6. MIGRATING JMS-ADMINISTERED OBJECTS AND BRIDGES

HornetQ creates and deploys JMS connection factories, destinations, and bridges differently to JBoss Messaging.

In JBoss Messaging, JMS objects and bridges are configured as MBean services within the application server. In HornetQ, these are implemented as POJOs.

To migrate the configuration of these objects and bridges from JBoss Messaging to HornetQ, you must understand how parameters map from JBoss Messaging to HornetQ.

Table 2.3, “JMS Connection Factory Configuration Mappings” maps these parameters. Unless otherwise described, all HornetQ Object and Bridge attributes are specified in ***`JBOSS_DIST`*****`/jboss-as/server/`*****`<PROFILE>`*****`/deploy/hornetq/hornetq-jms.xml`**. Appendix A, *Configuration Reference* contains all supported directives for **`hornetq-jms.xml`**.

Table 2.3. JMS Connection Factory Configuration Mappings

JBoss Messaging ConnectionFactory Attributes	HornetQ JMS ConnectionFactory Attributes
`ClientID`	**`connection-factory.client-id`**
`JNDIBindings`	**`connection-factory.entries`**
`PrefetchSize`	**`connection-factory.consumer-window-size`**
`SlowConsumers`	N/A - equivalent to **`consumer-window-size=0`**
`StrictTck`	N/A
`SendAcksAsync`	**`connection-factory.block-on-acknowledge`**

JBoss Messaging ConnectionFactory Attributes	HornetQ JMS ConnectionFactory Attributes
`DefaultTempQueueFullSize`, **`DefaultTempQueuePageSize`**, **`DefaultTempQueueDownCacheSize`**	N/A
`DupsOKBatchSize`	**`connection-factory.dups-ok-batch-size`**
`SupportsLoadBalancing`	N/A
`SupportsFailover`	N/A
`DisableRemotingChecks`	N/A
`LoadBalancingFactory`	**`connection-factory.connection-load-balancing-policy-class-name`**
`Connector`	**`connection-factory.connectors`**
`EnableOrderingGroup`, **`DefaultOrderingGroup`**	N/A

Table 2.4, “JMS Queue Configuration Mappings” describes how JBoss Messaging Queue attributes map to HornetQ JMS Queue attributes. Unless otherwise specified, these attributes are defined in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**. If not specified in **`hornetq-configuration.xml`**, they are specified in **`JBOSS_DIST/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-jms.xml`**.

Table 2.4. JMS Queue Configuration Mappings

JBoss Messaging Queue Attributes	HornetQ JMS Queue Attributes
`Name`	**`queue.name`** - defined in **`hornetq-jms.xml`**
`JNDIName`	**`queue.entry`** - defined in **`hornetq-jms.xml`**
`DLQ`	**`address-settings.dead-letter-address`**
`ExpiryQueue`	**`address-settings.expiry-address`**
`RedeliveryDelay`	**`address-settings.redelivery-delay`**
`MaxDeliveryAttempts`	**`address-settings.max-delivery-attempts`**

JBoss Messaging Queue Attributes	HornetQ JMS Queue Attributes
`SecurityConfig`	`security-settings`
`FullSize`	`address-settings.max-size-bytes` - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
`PageSize`	`address-settings.page-size-bytes` - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
`DownCacheSize`	Not Supported
`CreatedProgrammatically`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`MessageCount`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`ScheduledMessageCount`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`MessageCounter`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`MessageCounterStatistics`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`ConsumerCount`	Refer to `org.hornetq.api.jms.management.JMSQueueControl` to retrieve this attribute.
`DropOldMessageOnRedeploy`	Not Supported
`MaxSize`	Not Supported
`Clustered`	Not Supported

Table 2.5, “JMS Topic Configuration Mappings”`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml` describes how JBoss Messaging Topic attributes map to HornetQ JMS Topic attributes. Unless otherwise specified, these

attributes are defined in ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml**.

Table 2.5. JMS Topic Configuration Mappings

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
`Name`	`topic.name` - defined in `hornetq-jms.xml`
`JNDIName`	`topic.entry` - defined in `hornetq-jms.xml`
`DLQ`	`address-settings.dead-letter-address`
`ExpiryQueue`	`address-settings.expiry-address`
`RedeliveryDelay`	`address-settings.redelivery-delay`
`MaxDeliveryAttempts`	`address-settings.max-delivery-attempts`
`SecurityConfig`	`security-settings`
`FullSize`	`address-settings.max-size-bytes` - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
`PageSize`	`address-settings.page-size-bytes` - HornetQ paging attributes do not exactly match JBoss Messaging paging attributes. Refer to the appropriate chapter for details.
`DownCacheSize`	N/A
`CreatedProgrammatically`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.
`MessageCounterHistoryDayLimit`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.
`MessageCounters`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.
`AllMessageCount`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.
`DurableMessageCount`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.
`NonDurableMessageCount`	Refer to `org.hornetq.api.jms.management.TopicControl` to retrieve this attribute.

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
`AllSubscriptionsCount`	Refer to **`org.hornetq.api.jms.management.TopicControl`** to retrieve this attribute.
`DurableSubscriptionsCount`	Refer to **`org.hornetq.api.jms.management.TopicControl`** to retrieve this attribute.
`NonDurableSubscriptionsCount`	Refer to **`org.hornetq.api.jms.management.TopicControl`** to retrieve this attribute.
`MaxSize`	N/A
`Clustered`	N/A
`DropOldMessageOnRedeploy`	N/A

The table below shows how JBoss Messaging Bridge attributes map to HornetQ JMS Bridge attributes. HornetQ's JMS Bridge attributes are defined in its Bean configuration files. Refer to Chapter 31, *The JMS Bridge* for further details.

Table 2.6. JMS Bridge Configuration Mappings

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
`SourceProviderLoader`	**`SourceCFF`**
`TargetProviderLoader`	**`TargetCFF`**
`SourceDestinationLookup`	**`SourceDestinationFactory`**
`TargetDestinationLookup`	**`TargetDestinationFactory`**
`SourceUsername`	Source user name parameter
`SourcePassword`	Source user password parameter
`TargetUsername`	Target user name parameter
`TargetPassword`	Target password parameter
`QualityOfServiceMode`	Quality of Service parameter

JBoss Messaging Topic Attributes	HornetQ JMS Topic Attributes
`Selector`	Selector parameter
`MaxBatchSize`	`Max batch size parameter`
`MaxBatchTime`	`Max batch time parameter`
`SubName`	`Subscription name parameter`
`ClientID`	`Client ID parameter`
`FailureRetryInterval`	`Failure retry interval parameter`
`MaxRetries`	`Max retry times parameter`
`AddMessageIDInHeader`	`Add Message ID in Header parameter`

2.7. OTHER CONFIGURATION IN JBOSS MESSAGING

There are two kinds of configurations for JBoss Messaging dependent services, one for JBoss Remoting and the other for JGroups. Details of this configuration is outside the scope of this document. Consult the JBoss Remoting and JGroups documentation for these details.

HornetQ has its own pluggable transportation architecture and clustering implementation, and currently uses Netty as its transport.

2.8. MIGRATING EXISTING MESSAGES

Once you have migrated all JMS destinations to HornetQ, you can migrate existing messages. The JMS Bridge can be used to move existing messages from JBoss Messaging to HornetQ. Any prepared transactions should be completed with JBoss Messaging.

2.9. APPLICATIONS THAT USE MANAGEMENT APIS

JBoss Messaging exposes its management API through MBean interfaces. HornetQ includes a number of different management APIs (see Chapter 28, *Management* for details).

While JBoss Messaging management APIs are accessed via JMX, HornetQ provides access to its management APIs through:

JMX

JMX is the standard method of managing Java applications.

Core API

Management operations are sent to the HornetQ server using Core messages.

JMS API

Management operations are sent to the HornetQ server using JMS messages

The same functionality can be achieved through all three management methods.

The following table lists the JBoss Messaging management Objects alongside their HornetQ JMX counterparts. For the other management APIs available in HornetQ, see Chapter 28, *Management*.

Table 2.7. JBoss Messaging and HornetQ Management Object API Mappings

org.jboss.jms.server. Class	org.hornetq.api.jms.management. Class
`ServerPeer`	`JMSServerControl`
`connectionfactory.ConnectionFactory`	`ConnectionFactoryControl`
`destination.QueueService`	`JMSQueueControl`
`destination.TopicService`	`TopicControl`

NOTE

Some JBoss Messaging MBeans have no equivalent in HornetQ; for example, there is no HornetQ equivalent for `JDBCPersistenceManagerService` because HornetQ does not require a datasource as JBoss Messaging does.

CHAPTER 3. MESSAGING CONCEPTS

HornetQ is an asynchronous messaging system; an example of Message Oriented Middleware, which will be referred to as *messaging systems*.

3.1. MESSAGING CONCEPTS

Messaging systems are designed to loosely couple heterogeneous systems together, while maintaining reliability, transactions, and many other features.

Unlike systems based on a 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, however this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end-to-end provides improvements to hardware resource usage, minimizes the number of threads blocking IO operations, and uses 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. Message senders and consumers are completely independent, which allows flexible, loosely coupled systems to be created.

Large enterprises often 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 (ESB). Using a message bus to decouple disparate systems can allow the system to grow and adapt more easily, or retire obsolete systems.

3.2. MESSAGING STYLES

Messaging systems normally support two main styles of asynchronous messaging: message queue messaging (also known as point-to-point messaging) and publish subscribe messaging. They are summarized briefly here:

3.2.1. The Point-To-Point 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. Some time later the messaging system delivers the message to a consumer. The consumer processes the message and acknowledges the message when it is done. 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 acknowledgment from the consumer, the message will be available to be delivered to a consumer again, upon recovery.

With point-to-point messaging, there can be many consumers in the queue but a particular message will only ever be consumed by one of them. Senders (also known as producers) to the queue are completely decoupled from receivers (also known as consumers) of the queue; that is, they do not know of each other's 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 us 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 us 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 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 are not 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 the order processor updates the order database, 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 acknowledgment will be completed in a single transaction to conform with atomicity, consistency, isolation, durability(ACID) properties.

3.2.2. The Publish-Subscribe Pattern

With publish-subscribe messaging, many senders can send messages to an entity on the server, often called a topic (it is used this way in the JMS world, for example).

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.

3.3. 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 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; you do not want your orders fulfilled more than once or any of your orders to be lost, for example.

In other cases you may not care about a once only 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.

3.4. TRANSACTIONS

HornetQ supports sending and acknowledging messages as part of a large global transaction by using the Java mapping of `XA: JTA`.

3.5. 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 does not need to survive a restart.

3.6. MESSAGING APIS AND PROTOCOLS

Many messaging systems provide their own proprietary APIs, which the client can use to communicate with the messaging system.

There are also some standard ways of operating with messaging systems and some emerging standards in this space. Some of these are explored in the next section.

3.6.1. Java Message Service (JMS)

JMS is part of Sun's Java EE specification. It is a Java API that encapsulates both message queue and publish-subscribe messaging patterns. JMS is a lowest common denominator specification. That is, it was created to encapsulate common functionality of the messaging systems that already existed and 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 running 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.

3.6.2. System specific APIs

Many systems provide their own programmatic API to interact with the messaging system. The advantage of this is that it allows the full set of system functionality to be exposed to the client application. APIs 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 beyond that accessible via the JMS API.

3.7. 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 messaging systems.

HornetQ provides automatic fail-over; where your sessions are reconnected to the backup server in the event of live server failure. Your applications must support High Availability correctly, according to the requirements of JMS.

For more information on HA and how applications must support this mode, refer to Section 2.3, “Client-side Failure Handling”.

3.8. CLUSTERS

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

Cluster support can vary between messaging systems, with some systems having fairly basic clusters with the cluster members being hardly aware of each other.

HornetQ provides a very configurable 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 can automatically redistribute messages between nodes of a cluster to prevent message loss on any particular node.

For full details on clustering, refer to Chapter 36, *Clusters*.

3.9. 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 messages from a queue on one server and routes messages to another queue on a different server. Bridges cope with unreliable connections, automatically reconnecting when the connection is 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 refer to Chapter 34, *Core Bridges* and Chapter 33, *Diverting and Splitting Message Flows*.

CHAPTER 4. CORE ARCHITECTURE

HornetQ core is designed as a set of Plain Old Java Objects (POJOs). It has also been designed to have as few dependencies on external jars as possible. As a result HornetQ core has only one more jar dependency than the standard JDK classes: **`netty.jar`**. This is because some of the netty buffer classes are used internally.

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

Using a high performance journal allows persistence message performance, which is 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:

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.

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 associate with JMS and does not know anything about JMS. It is 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 described in Figure 4.1, “HornetQ Application Interaction Schematic”.

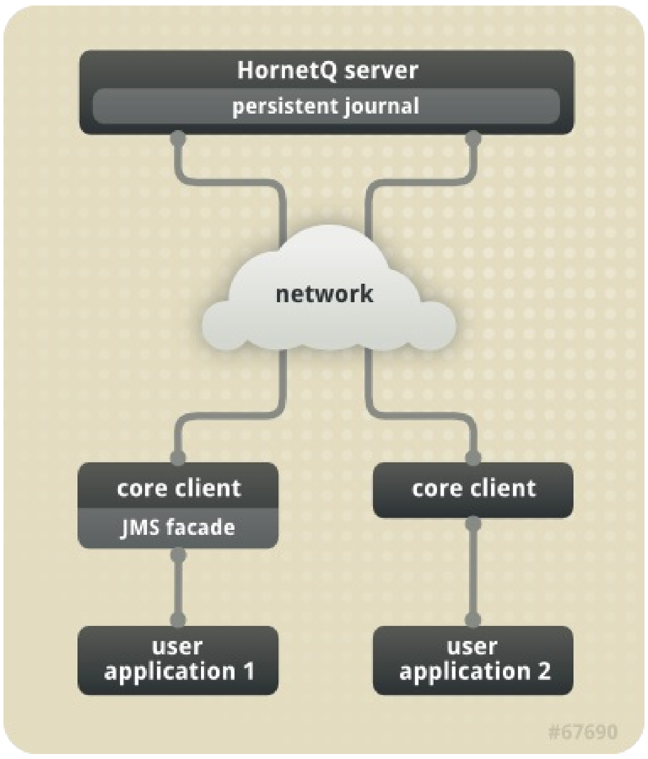


Figure 4.1. HornetQ Application Interaction Schematic

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.

CHAPTER 5. USING THE SERVER

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

5.1. LIBRARY PATH

If you are 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 do not specify **`java.library.path`** at your Java options then the JVM will use the environment variable **`LD_LIBRARY_PATH`**.

5.2. SYSTEM PROPERTIES

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

For more information on configuring logging, refer to Chapter 41, *Logging*.

5.3. CONFIGURATION FILES

The configuration files are stored in a number of locations in the JBoss Enterprise Application Server directory structure. In all cases, you must change the file in each server profile you want to run because they are not shared between profiles.

Files located in /deploy/hornetq/

hornetq-configuration.xml

This is the main HornetQ configuration file. All the parameters in this file are described in Appendix A, *Configuration Reference*. Refer to Section 5.4, “The Main Configuration File” for more information on this file.

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. By default, this is set to true.

hornetq-jboss-beans.xml

This is the JBoss Microcontainer beans file which defines what beans the Microcontainer should create and what dependencies to enforce between them.

hornetq-jms.xml

The distribution configuration by default includes a server side JMS service which mainly deploys JMS Queues, Topics and Connection Factories from this file into JNDI. If you are not using JMS, or you do not need to deploy JMS objects on the server side, then you do not need this file. For more information on using JMS, refer to Chapter 6, *Using JMS*.

Files located in /conf/props/

hornetq-users.properties

HornetQ ships with a security manager implementation that obtains user credentials from the **hornetq-users.properties** file. This file contains user and password information. For more information on security, refer to Chapter 29, *Security*.

hornetq-roles.properties

This file contains user names defined in **hornetq-users.properties** with the roles they have permission to use. For more information on security, refer to Chapter 29, *Security*.

It is also possible to use system property substitution in all the configuration files in a server profile 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.core.remoting.impl.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 two values have been replaced 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. That is, **${hornetq.remoting.netty.host}**, however the system property *must* be supplied in that case.

5.4. THE MAIN CONFIGURATION FILE

The configuration for the HornetQ core server is contained in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**. This is what the **FileConfiguration** bean uses to configure the messaging server.

There are many attributes which you can configure in HornetQ, however in most cases the defaults are sufficient for basic operation.

Every attribute has a default setting, therefore a file with a single empty <configuration> element is a valid configuration file. Non-default configuration is explained throughout the manual, or you can refer to Appendix A, *Configuration Reference* to access all elements in a reference-style format.

CHAPTER 6. 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.

This section will cover the main steps in configuring the server for JMS and creating a simple JMS program. It will also show how to configure and use JNDI, and how to use JMS with HornetQ without using any JNDI.

6.1. A SIMPLE ORDERING SYSTEM - CONFIGURATION EXAMPLE

This configuration example uses a single JMS Queue called **`OrderQueue`**, with a single **`MessageProducer`** sending an order messages to the queue. A single **`MessageConsumer`** consumes the order message from the queue.

The queue is configured to be **`durable`**, (it will survive a server restart or crash).

The example also shows how to specify the queue in the server JMS configuration so it is created automatically without having to explicitly create it from the client.

6.1.1. JMS Server Configuration

The file **`hornetq-jms.xml`** on the server classpath (in standard configurations, ***`JBOSS_DIST`*****`/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-jms.xml`**) 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.

The example will 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 ConnectionFactory instances you can deploy from the file. Here is the 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="NettyConnectionFactory">
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
      </entries>
   </connection-factory>
   <queue name="OrderQueue">
      <entry name="queues/OrderQueue"/>
   </queue>
</configuration>
```

One ConnectionFactory called **`ConnectionFactory`** is deployed and bound in just one place in JNDI as given by the **`entry`** element. ConnectionFactory instances can be bound in many places in JNDI if it is required.

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 ***`<JBOSS_HOME>`*`/jboss-as/server/`*`<PROFILE>`*`/deploy/hornetq/hornetq-configuration.xml`**, which defines the transport and parameters used to actually connect to the server.

6.1.2. Connection Factory Types

The JMS API doc provides several connection factories for applications. HornetQ JMS users can choose to configure the types for their connection factories. Each connection factory has a **`signature`** attribute and a **`xa`** parameter, the combination of which determines the type of the factory.

Attribute **`signature`** has three possible string values (*generic*, *queue* and *topic*).

`xa` is a boolean type parameter. The following table gives their configuration values for different connection factory interfaces.

Table 6.1. Configuration for Connection Factory Types

signature	xa	Connection Factory Type
generic (default)	false (default)	javax.jms.ConnectionFactory
generic	true	javax.jms.XAConnectionFactory
queue	false	javax.jms.QueueConnectionFactory
queue	true	javax.jms.XAQueueConnectionFactory
topic	false	javax.jms.TopicConnectionFactory
topic	true	javax.jms.XATopicConnectionFactory

As an example, the following configures an XAQueueConnectionFactory:

```
<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" signature="queue">
        <xa>true</xa>
        <connectors>
```

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

6.1.3. The code

The code for the example is available below.

The first step is to create a JNDI initial context from which to look up JMS objects.

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

The next step is to look up the connection factory:

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

Followed by looking up the Queue:

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

Next, create a JMS connection using the connection factory:

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

Create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

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

Create a MessageProducer that will send orders to the queue:

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

Create a MessageConsumer which will consume orders from the queue:

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

Make sure to start the connection, or delivery will not occur on it:

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

Create a simple TextMessage and send it:

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

Consume the message:

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

WARNING

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

It is 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 Chapter 43, *Performance Tuning*.

6.2. DIRECTLY INSTANTIATING JMS RESOURCES WITHOUT USING JNDI

It is a very common usage pattern to look up JMS *Administered Objects* (that is JMS queue, topic and ConnectionFactory instances) from JNDI. However in some cases, a JNDI server is not available, and using JMS is still required, or it is preferable to directly instantiate objects. This is possible with HornetQ, which supports the direct instantiation of JMS queue, topic and ConnectionFactory instances.

The following is a simple example, which does not use JNDI at all:

Create the JMS ConnectionFactory object via the HornetQJMSClient Utility class. Note you need to provide connection parameters and specify which transport you are using. For more information on connectors refer to Chapter 14, *Configuring the Transport*.

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

Also create the JMS Queue object via the HornetQJMSClient Utility class:

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

Next create a JMS connection using the connection factory:

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

Create a non transacted JMS Session, with AUTO_ACKNOWLEDGE acknowledge mode:

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

Create a MessageProducer that will send orders to the queue:

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

Create a MessageConsumer which will consume orders from the queue:

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

Make sure you start the connection, or delivery will not occur on it:

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

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());
```

6.3. SETTING THE CLIENT ID

The client ID for a JMS client is needed to create durable subscriptions. It is possible to configure this on the connection factory in ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml**, and can be set via the <client-id> directive. Any connection created by this connection factory will have this set as its client ID.

6.4. 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 acknowledgments 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 (Mebibyte).

6.5. SETTING THE TRANSACTION BATCH SIZE

When receiving messages in a transaction it is possible to configure the consumer to send acknowledgments in batches rather than individually saving valuable bandwidth. This can be configured on the connection factory via the **transaction-batch-size** element. The default is 1024 * 1024 (bytes).

CHAPTER 7. USING CORE

HornetQ core is a completely JMS-agnostic messaging system with its own non-JMS API. This is called the *core API*.

You can use the core API directly if you do not want to use JMS. 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.

7.1. CORE MESSAGING CONCEPTS

Some of the core messaging concepts are similar to JMS concepts, but there are still differences between them. In general, the core messaging API is simpler than the JMS API, since distinctions between queues, topics and subscriptions are removed. Each of these major core messaging concepts will be discussed in turn.

7.1.1. Message

- A message is the unit of data that 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 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 not 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 have an optional time stamp which represents the time the message was sent.
- HornetQ also supports the sending or consuming of very large messages - much larger than can fit in available RAM at any one time.

7.1.2. 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 be 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.

7.1.3. Queue

Queues can be durable, which means 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.

7.1.4. 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.

7.1.5. 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 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 feature ensures 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 acknowledgments asynchronously, you can create true end-to-end asynchronous systems which is not possible using the standard JMS API. For more information on this feature refer to Chapter 18, *Guarantees of sends and commits*.

7.1.6. 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.

7.1.7. 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

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

It is 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 suffer from poor performance. This is discussed further in Chapter 43, *Performance Tuning*.

7.2. SIMPLE CORE EXAMPLE

Here is 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();
```

CHAPTER 8. 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. For example, 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. For example the JMS Topic with name "news.europe" would map to the core address "jms.topic.news.europe"

For example, 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>
```

CHAPTER 9. THE CLIENT CLASSPATH

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

WARNING

All the jars mentioned here can be found in the ***$JBOSS_HOME*/client** 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.

9.1. HORNETQ CORE CLIENT

If you are using just a pure HornetQ Core client (for example, no JMS) then you need **hornetq-core-client.jar** and **netty.jar** on your client classpath.

9.2. JMS CLIENT

If you are using JMS on the client side, then include also **hornetq-jms-client.jar** and **jboss-javaee.jar**.

NOTE

jboss-javaee.jar only 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.

9.3. JMS CLIENT WITH JNDI

If you are looking up JMS resources from the JNDI server, you will also need the jar **jnp-client.jar** jar on your client classpath as well as any other jars mentioned previously.

CHAPTER 10. ROUTING MESSAGES WITH WILD CARDS

HornetQ allows the routing of messages via wildcard addresses.

If a queue is created with an address of **`queue.news.#`**, for example, then it will receive any messages sent to addresses that match this. Take these, for example: **`queue.news.europe`** or **`queue.news.usa`** or **`queue.news.usa.sport`**. If you create a consumer on this queue, this allows a consumer to consume messages which are sent to a *hierarchy* of addresses.

NOTE

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

To enable this functionality set the property **`wild-card-routing-enabled`** in the **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** file to **`true`**. This is **`true`** by default.

For more information on the wild card syntax take a look at Chapter 11, *Understanding the HornetQ Wildcard Syntax* chapter.

CHAPTER 11. UNDERSTANDING THE HORNETQ WILDCARD SYNTAX

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

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' or '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'.

CHAPTER 12. 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.

Filter expressions are used in several places in HornetQ.

- Predefined Queues. When pre-defining a queue, either in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** or **`JBOSS_DIST/jboss-as/server/<PROFILE>/deploy/hornetq/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 34, *Core Bridges*).
- Diverts can be defined with an optional filter expression. Only matching messages will be diverted (see Chapter 33, *Diverting and Splitting Message Flows*).
- Filters are also used programmatically when creating consumers, queues and in several places as described in Chapter 28, *Management*.

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 expression 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. For example, **`HQPriority = 3 and department = 'payroll'`**. This refers to a message with a priority of three and a department of 'payroll'.

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 time stamp 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.

CHAPTER 13. PERSISTENCE

This chapter covers persistence and its configuration in HornetQ.

HornetQ handles its own persistence. It ships with a high-performance journal, which is optimized for messaging-specific use cases.

The HornetQ journal is *append only* with a configurable file size, which improves performance by enabling single write operations. It consists of a set of files on disk, which are initially pre-created to a fixed size and filled with padding. As server operations (add message, delete message, update message, etc.) are performed, records of the operations are appended to the journal until the journal file is full, at which point the next journal file is used.

A sophisticated garbage collection algorithm determines whether journal files can be reclaimed and re-used when all of their data has been deleted. A compaction algorithm removes dead space from journal files and compresses the data.

The journal also fully supports both local and XA transactions.

The majority of the journal is written in Java, but interaction with the file system has been abstracted to allow different pluggable implementations. The two implementations shipped with HornetQ are:

Java Non-blocking IO (NIO)

Uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform with a Java 6 or later runtime.

Linux Asynchronous IO (AIO)

Uses a native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, HornetQ receives a message when data has been persisted. This removes the need for explicit syncs. AIO will typically provide better performance than Java NIO, but requires Linux kernel 2.6 or later and **libaio**.

AIO also requires **`ext2`**, **`ext3`**, **`ext4`**, **`jfs`** or **`xfs`** type file systems. On NFS, AIO falls back to slower, synchronous behavior.

NOTE

On Red Hat Enterprise Linux, install **libaio** with the following command:

```
yum install libaio
```

The standard HornetQ core server uses the following journal instances:

bindings journal

Stores bindings-related data, including the set of queues 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 typically has low throughput in comparison to the message journal.

The files on this journal are prefixed as **`hornetq-bindings`**. Each file has a **`bindings`** extension. File size is **`1048576`** bytes, and it is located in the bindings folder.

JMS journal

Stores all JMS-related data, for example, any JMS queues, topics or connection factories and any

JNDI bindings for these resources. Any JMS resources created with the management API are persisted to this journal. Any resources configured with configuration files are not. This journal is created only if JMS is in use.

message journal

Stores all message-related data, including messages themselves and ***`duplicate-id`*** caches. By default, HornetQ uses AIO for this journal. If AIO is not available, it will automatically fall back to NIO.

Large messages are persisted outside the message journal. For more information see Chapter 21, *Large Messages*.

In low memory situations, configure HornetQ to page messages to disk. See Chapter 22, *Paging* for more information.

If persistence is not required, HornetQ can be configured not to persist any data, as discussed in Section 13.4, “Configuring HornetQ for Zero Persistence”.

13.1. CONFIGURING THE BINDINGS JOURNAL

The bindings journal is configured using the following attributes in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**.

`bindings-directory`

The location of the bindings journal. The default value is **`data/bindings`**.

`create-bindings-dir`

If **`true`**, and the bindings directory does not exist, the bindings directory is created automatically at the location specified in **`bindings-directory`**. The default value is **`true`**.

13.2. CONFIGURING THE JMS JOURNAL

The JMS journal shares its configuration with the bindings journal.

13.3. CONFIGURING THE MESSAGE JOURNAL

The message journal is configured using the following attributes in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**.

`journal-directory`

The location of the message journal. The default value is **`data/journal`**. For best performance, this journal should be located on its own physical volume to minimize disk head movement. If this journal is stored on a storage area network, each journal instance on the network should have its own logical unit.

`create-journal-dir`

If **`true`**, the journal directory is created at the location specified in **`journal-directory`**. The default value is **`true`**.

`journal-type`

Valid values are **`NIO`** or **`ASYNCIO`**. If **`NIO`**, the Java NIO journal is used. If **`ASYNCIO`**, Linux asynchronous IO is used. If **`ASYNCIO`** is set on a non-Linux or non-libaio system, HornetQ detects this and falls back to **`NIO`**.

`journal-sync-transactional`

If **`true`**, HornetQ ensures all transaction data is flushed to disk on transaction boundaries (commit, prepare, and rollback). The default is **`true`**.

`journal-sync-non-transactional`

If **`true`**, HornetQ ensures non-transactional message data (sends and acknowledgments) are flushed to disk. The default is **`true`**.

`journal-file-size`

The size of each journal file in bytes. The default value is **`10485760`** bytes (10 megabytes).

`journal-min-files`

The minimum number of files the journal maintains. When HornetQ starts and there is no initial data, HornetQ pre-creates this number of files. Creating and padding journal files is an expensive operation, so to be avoided at run-time as files are filled. Pre-creating files means that as one is filled the journal can immediately resume with the next file without pausing to create it.

`journal-max-io`

The maximum number of write requests to hold in the IO queue. Write requests are queued here before being submitted to the system for execution. If the queue fills, writes are blocked until space becomes available in the queue. For NIO, this must be **`1`**. For AIO, this should be **`500`**. A different default value is maintained depending on whether NIO or AIO is used (**`1`** for NIO, **`500`** for AIO). The total max AIO must not be higher than what is configured at the operating system level (**`/proc/sys/fs/aio-max-nr`**), generally at 65536.

`journal-buffer-timeout`

HornetQ maintains a buffer of flush requests, and flushes the entire buffer either when it is full or when this timeout expires - whichever is soonest. This is used for both NIO and AIO and allows improved scaling when many concurrent writes and flushes are required.

`journal-buffer-size`

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

`journal-compact-min-files`

The minimum number of files before the journal will be compacted. The default value is **`10`**.

`journal-compact-percentage`

When less than this percentage of a journal is considered live data compacting will occur. The default value is **`30`**. **`journal-compact-min-files`** must also be fulfilled before compacting.

WARNING

Most disks contain hardware write caches, which increase the apparent performance of a disk because writes are cached and lazily written to disk later.

Many systems ship with disk write cache enabled by default, so even after syncing from the operating system there is no guarantee that the data has been written to disk. If a failure occurs, critical data can still be lost.

Some systems have non-volatile or battery-backed write caches. These will not necessarily lose data in the event of failure, but testing is essential.

If your disk does not have these backups in place, and is not part of a redundant array (for example, RAID), ensure that disk write cache is **`disabled`**. This can have negative effects on performance, but ensures data integrity.

On Linux, inspect or change your disk's write cache settings with the **hdparm** tool for IDE disks, or **sdparm** or **sginfo** tools for SCSI or SATA disks.

On Windows, check or change settings by right-clicking on the disk and selecting **Properties**.

13.4. CONFIGURING HORNETQ FOR ZERO PERSISTENCE

To configure HornetQ to perform zero persistence, set the **`persistence-enabled`** parameter in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** to **`false`**.

IMPORTANT

Once this parameter is set to **`false`**, no persistence will occur. No bindings data, message data, large message data, duplicate ID caches, or paging data will be persisted.

13.5. IMPORT/EXPORT THE JOURNAL DATA

You may want to inspect the existent records on each one of the journals used by HornetQ, and you can use the export/import tool for that purpose. The export/import are classes located in the **`hornetq-core.jar`**, you can export the journal as a text file by using this command:

```
java -cp hornetq-core.jar org.hornetq.core.journal.impl.ExportJournal
<JournalDirectory> <JournalPrefix> <FileExtension> <FileSize> <FileOutput>
```

To import the file as binary data on the journal (Notice you also require **`netty.jar)`**:

```
java -cp hornetq-core.jar:netty.jar
org.hornetq.core.journal.impl.ImportJournal <JournalDirectory>
<JournalPrefix> <FileExtension> <FileSize> <FileInput>
```

- JournalDirectory: Use the configured folder for your selected folder. Example: ./hornetq/data/journal

- JournalPrefix: Use the prefix for your selected journal, as discussed
- FileExtension: Use the extension for your selected journal, as discussed
- FileSize: Use the size for your selected journal, as discussed
- FileOutput: text file that will contain the exported data

CHAPTER 14. CONFIGURING THE TRANSPORT

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

This chapter covers the concepts required to use and configure HornetQ transports.

14.1. UNDERSTANDING ACCEPTORS

Acceptors are defined in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** using the following directives.

```
<acceptor name="netty">
   <factory-class>
org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-class>
   <param key="host"  value="${jboss.bind.address:localhost}"/>
   <param key="port"  value="${hornetq.remoting.netty.port:5445}"/>
</acceptor>
```

Acceptors are always defined inside an **acceptors** element. Multiple acceptors can be defined in one **acceptors** element. There is 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.

The above example defines an **acceptor** that uses Netty to listen for connections on port **5445**.

The **acceptor** element contains a sub-element **factory-class** which defines the factory used to create acceptor instances. In this case Netty is used to listen for connections, so the Netty implementation of **AcceptorFactory** is being used. The **factory-class** element determines which pluggable transport listens.

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.

14.2. UNDERSTANDING CONNECTORS

Where 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. Acceptors and Connectors are defined in the **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** file:

```
<connector name="netty">
  <factory-
class>org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</factory-
class>
  <param key="host" value="${jboss.bind.address:localhost}"/>
  <param key="port" value="${hornetq.remoting.netty.port:5445}"/>
</connector>
```

Connectors can be defined inside a <connectors> element. Multiple connectors can be defined in the same <connectors> element. There is no upper limit to the number of connectors per server.

Although connectors are used by the client, they are defined on the server for a number of reasons:

- 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 these cases the server needs to know how to connect to other servers. This is defined by *connectors*.

- If JMS and the server-side JMS service are used to instantiate JMS ConnectionFactory instances and bind them in JNDI, the JMS service needs to know which server the **HornetQConnectionFactory** will create connections to at the connection factory's creation.

  This is defined by the <connector-ref> element in the ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml** file on the server side. The following snippet from a **hornetq-jms.xml** file shows a JMS connection factory that references the netty connector defined in the **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** file:

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

14.3. CONFIGURING THE TRANSPORT DIRECTLY FROM THE CLIENT SIDE

This section shows how to configure a core **ClientSessionFactory** to connect with a server.

Connectors are used indirectly when configuring a core **ClientSessionFactory** to talk to a server. In this case, it is unnecessary to define a connector in the server-side configuration. Instead, create the parameters and configure the connector factory to be used by **ClientSessionFactory**.

The following **ClientSessionFactory** connects directly to the acceptor defined previously in this chapter. It uses the standard Netty TCP transport, and will attempt to connect on port 5446 to localhost (the default).

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

  connectionParams.put(

org.hornetq.core.remoting.impl.netty.TransportConstants.PORT_PROP_NAME,
    5446
  );

TransportConfiguration transportConfiguration =
  new TransportConfiguration(
```

```
      "org.hornetq.core.remoting.impl.netty.NettyConnectorFactory",
      connectionParams
    );

ClientSessionFactory sessionFactory =
  HornetQClient.createClientSessionFactory(transportConfiguration);

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

For JMS, you can configure the JMS connection directly on the client side without defining a connector on the server side or a connection factory in ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml**:

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

connectionParams.put(
  org.hornetq.core.remoting.impl.netty.TransportConstants.PORT_PROP_NAME,
  5446
);

TransportConfiguration transportConfiguration =
  new TransportConfiguration(
    "org.hornetq.core.remoting.impl.netty.NettyConnectorFactory",
    connectionParams
  );

ConnectionFactory connectionFactory =
  HornetQJMSClient.createConnectionFactory(transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();
```

14.4. CONFIGURING THE NETTY TRANSPORT

HornetQ uses Netty, a high-performance, low-level network library.

The Netty transport can be configured to use old (blocking) Java IO, NIO (non-blocking), TCP sockets, SSL, HTTP or HTTPS. A servlet transport is also provided.

14.4.1. Configuring Netty TCP

Netty TCP is a simple unencrypted TCP socket-based transport. Netty TCP can be configured to use blocking Java Asynchronous IO (AIO) or non-blocking Java NIO (NIO). NIO is recommended on the server-side for concurrent connection scalability; however, AIO can provide better latency if many concurrent connections are not required.

WARNING

If you are running connections across an untrusted network, consider SSL or HTTPS instead.

IMPORTANT

If non-blocking messages are sent then there is a chance that these could arrive on the server after the calling thread has completed. This means that the security context has been cleared. If this is the case then messages will need to be sent as blocking messages.

With the Netty TCP transport all connections are initiated from the client side. This is useful in situations where firewall policies only allow connections to be initiated in one direction.

All valid Netty transport keys are defined in **`org.hornetq.core.remoting.impl.netty.TransportConstants`**. Most parameters can be used with acceptors and connectors. Some only work with acceptors. 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`** the older, blocking Java IO will be used. If handling many concurrent connections to the server is a requirement, the non-blocking Java NIO method is highly recommended. Java NIO does not maintain a thread per connection so can scale to many more concurrent connections than the older blocking IO method. If handling many concurrent connections is not required, slightly better performance might be gained by using older blocking IO method. The default value for this property is **`false`** on the server side and **`false`** on the client side.

`host`

The host name or IP address to connect to (for connectors) or listen on (for acceptors). The default value is **`localhost`**.

IMPORTANT

The default for this variable is **`localhost`**. This is not accessible from remote nodes and must be modified for the server to accept incoming connections.

Acceptors can be configured with multiple comma-delimited hosts or IP addresses. Multiple addresses are not valid for connectors. **`0.0.0.0`** specifies that all network interfaces of a host should be accepted.

`port`

Specifies the port to connect to (for connectors) or listen on (for acceptors). The default value is **`5445`**.

`tcp-no-delay`

If **`true`**, Nagle's algorithm is enabled. The default value is **`true`**.

`tcp-send-buffer-size`

Defines the size of the TCP send buffer in bytes. The default value is **`32768`** (32 kilobytes). TCP buffer size should be tuned according to the bandwidth and latency of your network. The buffer size in bytes should be equal to the bandwidth in bytes-per-second multiplied by the network round-trip-time (RTT) in seconds. RTT can be measured using the **ping** utility. For fast networks, you may wish to increase the buffer size from the default value.

`tcp-receive-buffer-size`

Defines the size of the TCP receive buffer in bytes. The default value is **`32768`** (32 kilobytes).

`batch-delay`

HornetQ can be configured to place write operations into batches for up to **`batch-delay`** milliseconds. This can increase overall throughput for very small messages, but does so at the expense of an increase in average latency for message transfer. The default value is **`0`** milliseconds.

`direct-deliver`

When a message arrives on the server and is delivered to consumers, by default the delivery occurs on a different thread to that in which the message arrived. This gives the best overall throughput and scalability, especially on multi-core machines. However, it also introduces additional latency due to the context switch required. For the lowest latency (and possible reduction of throughput), set **`direct-deliver`** to **`true`** (the default). For highest throughput, set to **`false`**.

`nio-remoting-threads`

When configured to use NIO, by default HornetQ uses three times the number of threads as cores (or hyper-threads) reported by **`Runtime.getRuntime().availableProcessors()`** to process incoming packets. **`nio-remoting-threads`** overrides this and defines the number of threads to use. The default is **`-1`**, which represents three times the value from **`Runtime.getRuntime().availableProcessors()`**.

14.4.2. 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).

Netty SSL uses the same properties as Netty TCP in addition to the following properties:

`ssl-enabled`

Set to **`true`** to enable SSL.

`key-store-path`

Defines the path to the SSL key store on the client that holds the client certificates.

`key-store-password`

Defines the password for the client certificate key store on the client.

`trust-store-path`

Defines the path to the trusted client certificate store on the server.

`trust-store-password`

Defines the password to the trusted client certificate store on the server.

14.4.3. Configuring Netty HTTP

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

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

`http-enabled`

Set to **`true`** to enable HTTP.

`http-client-idle-time`

The period of time (in milliseconds) 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`

The period of time in milliseconds the server should wait before sending an empty HTTP response to keep the connection alive.

`http-server-scan-period`

How often, in milliseconds, to scan for clients requiring responses.

`http-requires-session-id`

When **`true`**, the client waits to receive a session ID after the first call. This is used when the HTTP connector is connecting to the servlet acceptor (not recommended).

14.4.4. Configuring Netty Servlet

Netty Servlet transport allows HornetQ traffic to be tunneled over HTTP to a servlet running in a servlet engine, which redirects it to an in-virtual machine HornetQ server.

This differs from the Netty HTTP transport in that traffic is routed through a servlet engine which may already be serving web applications. This allows HornetQ to be used where corporate policies allow only a single web server to listen on an HTTP port.

Configuring a servlet engine for the Netty Servlet transport

1. Deploy the servlet. A web application using the servlet might have a **`web.xml`** file similar to the following:

```
<?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.xsd"
  version="2.4">
  <servlet>
    <servlet-name>HornetQServlet</servlet-name>
    <servlet-
class>org.jboss.netty.channel.socket.http.HttpTunnelingServlet</serv
let-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>
```

2. Add the **`netty-invm`** acceptor to the server-side configuration in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**:

```
<acceptors>
  <acceptor name="netty-invm">
    <factory-class>
      org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
    </factory-class>
    <param key="use-invm" value="true"/>
    <param key="host" value="org.hornetq"/>
  </acceptor>
</acceptors>
```

3. Define a connector for the client in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**:

```
<connectors>
  <connector name="netty-servlet">
    <factory-class>
      org.hornetq.core.remoting.impl.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>
```

Init Parameters

`endpoint`

Defines the netty acceptor to which the servlet forwards its packets. Matches the name of the **`host`** parameter.

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

The servlet transport can also be used over SSL by adding the following configuration to the connector:

```
<connector name="netty-servlet">
  <factory-class>
    org.hornetq.core.remoting.impl.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 keystore"/>
  <param key="key-store-password" value="keystore password"/>
</connector>
```

You will also have to configure the application server to use a Key Store. Edit the SSL/TLS connector configuration in **`server/default/deploy/jbossweb.sar/server.xml`** like so:

```
<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 key store and password. See the Servlet SSL example shipped with HornetQ for more detail.

CHAPTER 15. DETECTING DEAD CONNECTIONS

This chapter discusses Connection Time-to-Live (TTL), and explains how HornetQ deals with crashed clients and clients that have exited without cleanly closing their resources.

15.1. CLEANING UP DEAD CONNECTION RESOURCES ON THE SERVER

Before a HornetQ client application exits, its resources should be closed using a **`finally`** block.

Example 15.1. Well Behaved Core Client Application

Below is 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();
   }
}
```

Example 15.2. Well Behaved JMS Client Application

This is 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();
   }
}
```

If a client crashes, server side resources like sessions can be left hanging on the server. This can cause a resource leak over time and lead to the server running out of memory or other resources.

The requirement for cleaning up dead client resources is balanced with HornetQ client reconnection support. The network between the client and the server can fail and then reboot, allowing the client to reconnect. If the resources have been cleaned, the reboot will fail.

To ensure that resources are cleaned upon client crash, while allowing for reboot time, the **`connection TTL`** is configurable. Each **`ClientSessionFactory`** has a defined **`connection TTL`**. The TTL determines how long the server will keep a connection alive in the absence of any data arriving from the client.

The client automatically sends ping packets to prevent the server from closing it down. If the server does not receive any packets on a connection for the connection TTL time, then it will close all the sessions on the server that relate to that connection.

You can configure this functionality on the client side or the server side using the following methods:

- On the client side, specify the **`ConnectionTTL`** attribute on a **`HornetQConnectionFactory`** instance (**`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/jms-ra.rar/META-INF/ra.xml`**)
- On the server side where connection factory instances are being deployed directly into JNDI, specify the ***`connection-ttl`*** parameter for the **`<connection-factory>`** directive in the ***`JBOSS_DIST`*`/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-jms.xml`** file.

The default value for ***`ConnectionTTL`*** is 60000 (milliseconds). A value of **`-1`** for the ***`ConnectionTTL`*** attribute means the server will never time out the connection on the server side.

To prevent clients specifying their own connection TTL, a global value can be set on the server side that overrides all other values. To do this, specify the **`connection-ttl-override`** attribute in ***`JBOSS_DIST`*`/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-jms.xml`** file. The default value for **`connection-ttl-override`** is **`-1`**, which allows clients to set their own values for connection TTL.

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

It is important that all core client sessions and JMS connections are always closed explicitly in a **`finally`** block when they are finished.

If a session or connection is not closed in a **`finally`** block, 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 closing a ClientSession you left open. Please make sure you close
```

```
all ClientSessions
  explicitly before letting them go out of scope!
[Finalizer] 20:14:43,244 WARNING
[org.hornetq.core.client.impl.DelegatingSession]
The session you did not 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.

The log will also print the line of user code where the JMS connection/client session that did not close was created. This enables the error to be pinpointed and corrected appropriately.

15.2. DETECTING FAILURE FROM THE CLIENT SIDE.

The client pings the server to prevent the server cleaning dead resources. It also pings the server to detect if the server or network has failed.

As long as the client is receiving data from the server it will consider the connection to still be 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 fail-over, or call any **`SessionFailureListener`** instances (or **`ExceptionListener`** instances if you are using JMS) depending on how it has been configured.

For JMS, the check period is defined by the **`ClientFailureCheckPeriod`** attribute on a **`HornetQConnectionFactory`** instance. If JMS connection factory instances are being deployed directly into JNDI on the server side, it can be specified in the ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml** configuration file, using the parameter **`client-failure-check-period`**.

The default value for the client failure check period is **`30000`**ms (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 the connection TTL value to allow clients to reconnect in case of transitory failure.

15.3. CONFIGURING ASYNCHRONOUS CONNECTION EXECUTION

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

A thread from a thread pool can be used instead to handle some packets so that the remoting thread is not tied up for too long.

NOTE

Processing operations asynchronously on another thread adds a little more latency.

Most short running operations are handled on the remoting thread for performance reasons. Ensure the parameter **`async-connection-execution-enabled`** in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** is set to true to enable asynchronous connection execution.

CHAPTER 16. 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. Sometimes, a transaction will be started and then forgotten. If this happens, the transaction will sit indefinitely. If configured, HornetQ can scan for old transactions and rollback any expired transactions. The default lifespan of a transaction is five minutes (or 3000000 milliseconds). That is, any transactions older than five minutes are removed.

This timeout lifespan can be changed by editing the **`transaction-timeout`** property in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/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.

CHAPTER 17. FLOW CONTROL

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

17.1. 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. Messages can build up if the consumer cannot process messages as fast as they are being delivered and installed on the internal buffer. This can potentially lead to a lack of memory on the client if they cannot be processed in time.

17.1.1. Window-Based Flow Control

HornetQ consumers improve performance by buffering a certain number of messages in a client-side buffer before passing them to the client to consume.

To prevent a network round trip for every message, 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 *unbound* buffer
- **`0`** to not buffer any messages.
- **`>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, consider the two extremes:

Fast consumers

Fast consumers can process messages as fast as they consume them.

To allow fast consumers, set the **`consumer-window-size`** to -1. This will allow *unbound* 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 take 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 two consumers; one of which is very slow. Messages are delivered in a circular 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 waiting to be processed in the buffer of the slow consumer which prevents 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.

17.1.1.1. 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.

17.1.1.2. Using JMS

If JNDI is used to look up the connection factory, the consumer window size is configured in ***`JBOSS_DIST`*`/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/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.

17.1.2. Rate limited flow control

It is also possible to control the *rate* at which a consumer can consume messages. This 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`**.

17.1.2.1. 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.

17.1.2.2. Using JMS

If JNDI is used to look up the connection factory, the max rate can be configured in **_JBOSS_DIST_/jboss-as/server/_<PROFILE>_/deploy/hornetq/hornetq-jms.xml**:

```
<connection-factory name="NettyConnectionFactory">
   <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. If you had a slow rate limit and a high window based limit the internal buffer in the client would soon fill up with messages.

17.2. PRODUCER FLOW CONTROL

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

17.2.1. 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.

17.2.1.1. Using Core API

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

17.2.1.2. Using JMS

If JNDI is used to look up the connection factory, the producer window size can be configured in **_JBOSS_DIST_/jboss-as/server/_<PROFILE>_/deploy/hornetq/hornetq-jms.xml**:

```
<connection-factory name="NettyConnectionFactory">
   <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.

17.2.1.3. Blocking producer window based flow control

Normally the server will always give the same number of credits as has been requested. However, it is also possible to set a maximum size on any address. This then blocks how many credits can be sent to the address so that its memory cannot be exceeded.

For example, if there is a JMS queue called "myqueue", the maximum memory size could be set to 10 MB, and the server will control the number of credits sent to any producers which are sending any messages to myqueue. This means that the total messages in the queue never exceeds 10 MB.

When the address gets full, producers will block on the client side until more space frees up on the address; that is, until messages are consumed from the queue thus freeing up space for more messages to be sent. This is called blocking producer flow control.

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 23.3, “Configuring Queues Through Address Settings”) block for the address and specify **`max-size-bytes`** and **`address-full-policy`**

The address block applies to all queues registered to that address. That is, 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 will not exceed ***`max-size-bytes`***.

Here is an example:

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

The above example would set the max size of the JMS queue "exampleQueue" to be 100,000 bytes and would block any producers sending to that address to prevent that max size being exceeded.

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

17.2.2. Rate limited flow control

The rate a producer can emit messages can be limited, in units of messages per second. This means that the producer will never produce messages at a rate higher than what has been set.

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`**.

17.2.2.1. 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.

17.2.2.2. Using JMS

If JNDI is used to look up the ConnectionFactory, the max rate can be configured in ***`JBOSS_DIST`*`/jboss-as/server/`*`<PROFILE>`*`/deploy/hornetq/hornetq-jms.xml`**:

```
<connection-factory name="NettyConnectionFactory">
   <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.

CHAPTER 18. GUARANTEES OF SENDS AND COMMITS

18.1. GUARANTEES OF TRANSACTION COMPLETION

When committing or rolling back a transaction with HornetQ, the request to commit or rollback is sent to the server. The call will be blocked 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 the 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 **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**.

18.2. GUARANTEES OF NON TRANSACTIONAL MESSAGE SENDS

If messages are sent to a server using a non-transacted session, HornetQ can block sending 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 be blocked until the message has reached the server, and a response has been sent back. The default value is **`false`**.

Setting the send block 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 it is recommended that you either batch many message sends together in a transaction (since with a transactional session, only the commit/rollback does not block every send), or use the *asynchronous send acknowledgments feature* described in Section 18.4, “Asynchronous Send Acknowledgments”.

If you are using JMS and the JMS service on the server to load your JMS connection factory instances into JNDI then these parameters can be configured in ***`JBOSS_DIST`*`/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-jms.xml`** using the elements **`block-on-`**

`durable-send` and **`block-on-non-durable-send`**. If you are using JMS but not using JNDI then you can set these values directly on the **`HornetQConnectionFactory`** instance using the appropriate setter methods.

If you are 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`**.

18.3. GUARANTEES OF NON-TRANSACTIONAL ACKNOWLEDGMENTS

If you are acknowledging the delivery of a message on 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`**

18.4. ASYNCHRONOUS SEND ACKNOWLEDGMENTS

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 18.2, “Guarantees of Non Transactional Message Sends”, 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.

To remedy this, HornetQ provides a feature called *asynchronous send acknowledgments*. With this feature, HornetQ can be configured to send messages without blocking in one direction and asynchronously getting acknowledgment from the server that the messages were received in a separate stream. By decoupling the send from the acknowledgment 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 acknowledgments is determined by the confirmation-window-size parameter on the connection factory or client session factory. refer to Chapter 32, *Client Reconnection and Session Reattachment* for more info on this.

18.4.1. Asynchronous Send Acknowledgments

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

Send messages as normal using your **`ClientSession`**, and as messages reach the server, the server will send back an acknowledgment of the send asynchronously. HornetQ calls your handler's **`sendAcknowledged(ClientMessage message)`** method, passing in a reference to the message that was sent.

To enable asynchronous send acknowledgments, make sure **`confirmation-window-size`** is set to a positive integer value (specified in bytes). For example, 10485760 (10 Mebibytes) .

CHAPTER 19. MESSAGE REDELIVERY AND UNDELIVERED MESSAGES

Message delivery can be unsuccessful, for example, if the session used to consume a message is rolled back. An undelivered message returns to the queue ready to be redelivered. However, this means multiple unsuccessful deliveries are possible, so messages can remain in the queue, clogging the system.

There are two options for these undelivered messages:

Delayed Redelivery

Message delivery can be delayed to allow the client time to recover from transient failures and not overload its network or CPU resources.

Dead Letter Address

Configure a dead letter address, to which messages are sent after being determined undeliverable.

These options can be combined for maximum flexibility.

19.1. DELAYED REDELIVERY

Delaying redelivery can be useful for clients that regularly fail or roll back. Without delayed redelivery, the system can get into a "thrashing" state, where delivery fails and the client rolls back to attempt redelivery ad infinitum, consuming CPU and network resources.

19.1.1. 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 that many milliseconds before redelivering the messages. Redelivery delay is enabled by default and set to 60000 (1 minute).

Address wildcards can be used to configure redelivery delay for a set of addresses (see Chapter 11, *Understanding the HornetQ Wildcard Syntax*), so redelivery delay does not have to be specified for each address individually.

19.2. 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 one or more queues where they can later be perused by the system administrator for action to be taken.

HornetQ addresses can be assigned a dead letter address. Once the messages have been 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.

19.2.1. Configuring Dead Letter Addresses

The 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 be removed after **`max-delivery-attempts`** unsuccessful attempts.

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

A dead letter address can be set globally for a set of matching addresses, with **`max-delivery-attempts`** set 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 11, *Understanding the HornetQ Wildcard Syntax*).

19.2.2. 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.

19.3. DELIVERY COUNT PERSISTENCE

In normal use, HornetQ does not persist an updated delivery count until a message is rolled back (that is, 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*, imposing a significant performance penalty.

However, if the delivery count is not updated persistently before message delivery, in the event of a server crash, the delivery of some messages may not be reflected in the delivery count. Therefore, during the recovery phase, the server may deliver the message with **`redelivered`** set to **`false`** when it should be **`true`**.

Since this behavior breaks strict JMS semantics, delivery count can be persisted before message delivery in HornetQ. However, this is disabled by default for performance reasons. To enable this behavior, set **`persist-delivery-count-before-delivery`** to **`true`** in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**:

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

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

CHAPTER 20. MESSAGE EXPIRY

Messages can be set with an optional *time to live* (***`TimeToLive`***) when sending them.

HornetQ will not deliver a message to a consumer after its time to live has been exceeded. If the message has not been delivered before the time to live is reached, the server can discard it.

HornetQ addresses can be assigned an expiry address so when messages are expired, the addresses 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.

20.1. 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 you 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

20.2. CONFIGURING EXPIRY ADDRESSES

Expiry addresses 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, they are removed from the queue and dropped. Address wildcards can be used to configure expiry address for a set of addresses (see Chapter 11, *Understanding the HornetQ Wildcard Syntax*).

20.3. 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**.

`message-expiry-scan-period`

How often the queues will be scanned to detect expired messages (in milliseconds, default is 30000ms, set to **`-1`** to disable the reaper thread)

`message-expiry-thread-priority`

The reaper thread priority (it must be between zero and nine; nine being the highest priority. The default is three).

CHAPTER 21. LARGE MESSAGES

HornetQ supports sending and receiving of large 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.

To send a large message, the user can set an **InputStream** on a message body. When that message is sent, HornetQ will read the **InputStream**. For example, a **FileInputStream** could be used to send a large message from a large 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. When the time comes to deliver them to a consumer they are read back off the disk, also in fragments, and re-transmitted. 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 large 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.

21.1. CONFIGURING THE SERVER

Large messages are stored on a disk directory on the server side, as configured in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**.

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-messages-
directory>${jboss.server.data.dir}/${hornetq.data.dir:hornetq}/largemessag
es</large-messages-directory>

...

</configuration>
```

By default the large message directory is **data/large-messages**.

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

21.2. SETTING THE LIMITS

Any message larger 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.

21.2.1. Using Core API

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

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

Section 14.3, “Configuring the transport directly from the client side” provides more information on how to instantiate the session factory.

21.2.2. Using JMS

If JNDI is used to look up the connection factory, the minimum large message size is specified in ***`JBOSS_DIST`*`/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-jms.xml`**.

```
...
<connection-factory name="NettyConnectionFactory">
   <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`**.

21.3. 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 two 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 on your disk, but you could also send things such as:

- JDBC Blobs
- **`SocketInputStream`**
- Things recovered from **`HTTPRequests`**, and so on.

Anything that implements **`java.io.InputStream`** for sending messages, or **`java.io.OutputStream`** for receiving them can be used.

21.3.1. 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 21.1. 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
saveToOutputStream (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.saveToOutputStream(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);
...
```

21.3.2. Streaming over JMS

When using JMS, HornetQ maps the streaming methods on the core API (see Table 21.1, “org.hornetq.api.core.client.ClientMessage API”) 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 will not 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`**.

21.4. STREAMING ALTERNATIVE

If you choose not to use the **`InputStream`** or **`OutputStream`** capability of HornetQ, the data can still be accessed directly in an alternative fashion.

On the Core API, get the bytes of the body as described in Chapter 7, *Using Core*.

```
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
}
```

21.5. CACHE LARGE MESSAGES ON CLIENT

Large messages are transferred from server to client by streaming. The message is broken into smaller packets and more packets will be received as the message is read. It is because of this 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 will not be able to resend a large message in case of failure.

To solve this problem, you can enable the property **`cache-large-message-client`** in 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 in the connection factory used by the JMS Bridge if the JMS Bridge is being used for large messages.

CHAPTER 22. PAGING

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

In such a situation it is not possible to store all of the queues in memory at one time, so HornetQ transparently *pages* messages in and out of memory as they are needed. This allows 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.

22.1. 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.

22.2. CONFIGURATION

You can configure the location of the paging folder

Global paging parameters are specified in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/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>${jboss.server.data.dir}/hornetq/paging</paging-directory>
            <page-max-concurrent-io>5</page-max-concurrent-io>

            ...
```

`paging-directory`

This is where page files are stored. HornetQ will create one folder for each address under this configured location. The default for this is data/paging.

`page-max-concurrent-io`

The maximum number of concurrent reads the system can make on the paging files. This may be increased depending on the expected number of paged destinations and the limits on the storage infrastructure.

22.3. PAGING MODE

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

NOTE

Paging is done individually per address. Configuring a max-size-bytes for an address means each matching address will have a maximum size specified. Please note it *does not* mean that the total overall size of all matching addresses is limited to max-size-bytes.

22.3.1. Configuration

Configuration is done in the address settings, in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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 22.1. Paging Address Settings

Property Name	Description	Default
`max-size-bytes`	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-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

Property Name	Description	Default
page-max-cache-size	Specifies the number of page files kept in memory to optimize input/output cycles during paging navigation.	5

22.4. 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

22.5. 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. This prevents the memory being exhausted on the server.

Producers will automatically unblock and be able to continue sending when memory is freed up on the server.

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

IMPORTANT

Blocking is not recommended when using bridges because it is ignored and the message always transferred. In case of a bridge, either configure the destination to use paging instead of blocking, or make sure there is always a consumer at the target destination.

In the default configuration, all addresses are configured to block producers after 10 MB of data are in the address.

22.6. CAUTION WITH ADDRESSES WITH MULTIPLE QUEUES

When a message is routed to an address that has multiple queues bound to it; for example a JMS subscription, there is only one copy of the message in memory. Each queue only deals with a reference to this. This means that the memory is only freed up once all queues referencing the message have delivered it.

For example:

- An address has ten 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 nine queues are empty even though messages have been sent.

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

IMPORTANT

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, those messages will not be consumed until the messages in memory have been consumed.

HornetQ does not scan through page files on disk to locate matching messages. This is not the primary role of a messaging system. A relational database is recommended for implementations using selectors to select small subsets of messages in very large queues, because this functionality is similar to executing queries over tables in a relational database.

IMPORTANT

Do not set ***`page-size-bytes`*** (on the server) to a value lower than ***`ack-batch-size`*** (in the client) or your system may appear to hang.

Messages remain in server memory until they are acknowledged the server, therefore contributing to message sizes 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 a message is not acknowledged explicitly, it will be acknowledged according to the "ack-batch-size" setting in the client.

CHAPTER 23. QUEUE ATTRIBUTES

Queue attributes can be set in one of two ways; 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.

23.1. PREDEFINED QUEUES

Queues can be predefined through configuration at a core level or at a JMS level. First look at a JMS level.

JMS Level

The following shows a queue predefined in the ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/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 doing this at a JMS level a naming convention is followed so the actual name of the core queue will be **jms.queue.selectorQueue**.

The mandatory <entry> element configures the name used to bind the queue to JNDI. <queue> can contain multiple <entry> elements to bind the same queue to different names.

The optional <selector> element defines what JMS message selector the predefined queue will have. Only messages that match the selector will be added to the queue. When omitted from the <queue> the default value is **null**.

The optional <durable> element specifies whether the queue is persisted. When omitted from the <queue> the default value is **true**.

Core Level

A queue can be predefined at a core level in the **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** file.

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

This is very similar to the JMS configuration, with the following exceptions:

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 12, *Filter Expressions*), *not* the JMS selector syntax.

23.2. 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 the queue will be deleted once the session is disconnected.

Take a look at Chapter 28, *Management* for a description of the management API for creating queues.

23.3. CONFIGURING QUEUES THROUGH ADDRESS SETTINGS

There are some attributes that are defined against an address wildcard rather than a specific queue. Here is an example of an **`address-setting`** entry that would be found in the **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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>
```

Address settings allow you to provide a block of settings which will be applied against any addresses 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 in Chapter 11, *Understanding the HornetQ Wildcard Syntax*

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 times a canceled message can be redelivered before sending it to the **`dead-letter-address`**. A full explanation can be found in Section 19.2.1, “Configuring Dead Letter Addresses”.

`redelivery-delay`

Defines how long to wait before attempting redelivery of a canceled message. Refer to Section 19.1.1, “Configuring Delayed Redelivery”.

`expiry-address`

Defines where to send a message that has expired. Refer to Section 20.2, “Configuring Expiry Addresses”.

`last-value-queue`

Defines whether a queue only uses last values or not. Refer to Chapter 25, *Last-Value Queues*.

`max-size-bytes` and `page-size-bytes`

Are used to set paging on an address. This is explained in Chapter 22, *Paging*.

`redistribution-delay`

Defines how long to wait when the last consumer is closed on a queue before redistributing any messages. See Section 36.6, “Message Redistribution”.

`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 17, *Flow Control*, Chapter 22, *Paging*.

CHAPTER 24. SCHEDULED MESSAGES

Scheduled messages differ from normal messages in that they will not be delivered until a specified time in the future.

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

24.1. 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 positive **`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 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.

CHAPTER 25. 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.

A typical example for Last-Value queue is for stock market prices, where you are only interested in the latest value for a particular stock.

25.1. 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 11, *Understanding the HornetQ Wildcard Syntax*).

25.2. 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());
```

CHAPTER 26. MESSAGE GROUPING

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group id; that is, they have the 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), 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.

26.1. 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.

26.2. 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 **`HornetQConnectionFactory`** which will pick a random unique id. This can also be set in the **`*JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml`** file like this:

```
<connection-factory name="NettyConnectionFactory">
   <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** configuration file as follows:

```
<connection-factory name="NettyConnectionFactory">
   <connectors>
      <connector-ref connector-name="netty-connector"/>
   </connectors>
   <entries>
      <entry name="/ConnectionFactory"/>
   </entries>
   <group-id>Group-0</group-id>
</connection-factory>
```

26.3. CLUSTERED GROUPING

Using message groups in a cluster is a bit more complex because messages with a particular group ID can arrive on any node. Each node needs to know which group IDs 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 a grouping handler. Each node has its own grouping handler and when a message is sent with a group ID assigned, the handlers will decide between them which route the message should take.

There are two types of handlers; local and remote. Each cluster should choose one node to have a local grouping handler and all the other nodes should have remote handlers. The Local handler makes the decision regarding what route to use. The remote handlers converse with this. Here is a sample configuration for both types of handler. This should be configured in the ***<JBOSS_HOME>*/jboss-as/server/*<PROFILE>*/deploy/hornetq/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 type of addressing it uses (jms, or core). The timeout attribute refers to how long to wait for a decision to be made (in milliseconds); an exception will be thrown during the send if this timeout is reached. This ensures that strict ordering is kept.

NOTE

Refer to Chapter 36, *Clusters* for information about configuring clusters.

The decision about where a message should be routed is initially proposed by the node that receives the message. The node will pick a suitable route as per the normal clustered routing conditions (that is, circulate available queues, use a local queue first, and choose a queue that has a consumer). If the proposal is accepted 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 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, normal single-server message group semantics take over and the message is attached 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 sent messages will be not be delivered and an exception will be 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.

26.3.1. 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, that is, if you have three nodes, have three 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 successfully 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. This means that grouping can still occur on fail-over.

CHAPTER 27. PRE-ACKNOWLEDGE MODE

JMS specifies three acknowledgment 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 referred to in HornetQ as *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.

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-acknowledgment 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. This is because they are being acknowledged first on the server, not when you commit the transaction.

27.1. USING PRE_ACKNOWLEDGE

This can be configured in the ***`JBOSS_DIST`*****`/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-jms.xml`** file on the **`connection factory`** like this:

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

Alternatively, to use pre-acknowledgment 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-acknowledgment mode using the core API you can set it directly on the **`ClientSessionFactory`** instance using the setter method.

CHAPTER 28. MANAGEMENT

HornetQ has an extensive management API that allows a user to:

- Modify a server configuration;
- Create new resources (for example, JMS queues and topics);
- Inspect these resources (for example, how many messages are currently held in a queue);
- Interact with it (that is, to remove messages from a queue).

All the operations allow a client to *manage* HornetQ. It also allows clients to subscribe to management notifications.

There are three 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 three 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.

28.1. THE MANAGEMENT API

The management API is the same regardless of the way you invoke management operations. A Java interface describing what can be invoked exists for each type of managed resource.

HornetQ exposes its managed resources in two 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 operation depends whether JMX, core messages, or JMS messages are used.

NOTE

A few management operations requires a filter parameter to choose which messages are involved by the operation. Passing **`null`** or an empty string means that the management operation will be performed on *all messages*.

28.1.1. Core Management API

HornetQ defines a core management API to manage core resources. In summary:

28.1.1.1. 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()`**
- **`deployQueue()`**
- **`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 is resumed, it will begin delivering the queued messages.

Listing and closing remote connections

Client 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 the case of a server crash, some transactions may require manual intervention when the server restarts. 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 (for example, **`getVersion()`** method to retrieve the server's version, and so on.)

28.1.1.2. 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 with the queue with the **`getRoles()`** method.

28.1.1.3. Core Queue Management

The bulk of the core management APIs deal 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 (for example, to remove a single message) or a filter (for example, 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 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 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 28.6, “Message Counters”). 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 (for example, **`getFilter()`** to retrieve the queue's filter if it was created with one, **`isDurable()`** to know whether the queue is durable or not, and so on).

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 is resumed, it will begin delivering the queued messages.

28.1.1.4. Other Core Resources Management

HornetQ allows the user to start and stop its remote resources (acceptors, diverts, bridges, and so on) so that a server can be taken offline without stopping it completely (for example, if other management operations must be performed such as resolving heuristic transactions). These resources are:

Acceptors

Acceptors 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 parameters of the acceptors can be retrieved using the **`AcceptorControl`** attributes (see Section 14.1, “Understanding Acceptors”)

Diverts

Diverts 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 33, *Diverting and Splitting Message Flows*)

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 34, *Core Bridges*)

Broadcast groups

Broadcast groups 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 36.2.1, “Broadcast Groups”)

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 36.2.2, “Discovery Groups”)

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 36.3.1, “Configuring Cluster Connections”)

28.1.2. JMS Management API

HornetQ defines a JMS Management API to manage JMS administrated objects (that is, JMS queues, topics and connection factories).

28.1.2.1. 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 field input as a comma-separated list of key=value (that is, **`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 (seeChapter 14, *Configuring the Transport* 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()`**.

28.1.2.2. 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 (that is, **`getConsumerWindowSize()`** to retrieve the consumer window size for flow control, **`isBlockOnNonDurableSend()`** to know whether the producers created from the connection factory will block when sending non-durable messages, and so on).

28.1.2.3. 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 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 28.6, “Message Counters”)

Retrieving the queue attributes

The **`JMSQueueControl`** exposes JMS queue settings through its attributes (for example, **`isTemporary()`** to know whether the queue is temporary or not, **`isDurable()`** to know whether the queue is durable or not, and so on.)

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.

28.1.2.4. 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, and so on). 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 determine the number of messages held for a given subscription (with an optional message selector to determine the number of messages matching the selector).

28.2. USING MANAGEMENT VIA JMX

HornetQ can be managed using JMX.

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"
```

The MBean is:

```
org.hornetq.api.jms.management.JMSQueueControl
```

The MBean **`ObjectName`** is 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.

28.2.1. Configuring JMX

By default, JMX is enabled to manage HornetQ. It can be disabled by setting **`jmx-management-enabled`** to **`false`** in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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`**.

NOTE

Remote connections to JMX are not enabled by default for security reasons. System properties must be set in **`run.sh`** or **`run.bat`** scripts.

By default, the 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**:

```
<!-- use a specific JMX domain for HornetQ MBeans -->
<jmx-domain>my.org.hornetq</jmx-domain>
```

28.3. USING MANAGEMENT VIA CORE API

The core management API in HornetQ is called by sending core messages to 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 a management message is sent to the management address, HornetQ processes the message in the following way:

- extracts the information
- invokes the operation on the managed resources
- sends a *management reply* to the management message's reply-to address.

The management reply sent to the reply-to address is controlled by the **`org.hornetq.core.client.impl.ClientMessageImpl.REPLYTO_HEADER_NAME`** parameter.

A **`ClientConsumer`** can be used to consume the management reply and retrieve the result of the operation (if any) stored in the body of the reply. For portability, results are returned as a JSON 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:

Procedure 28.1. Invoking Management Operations

1. **Step One**
   Create a **`ClientRequestor`** to send messages to the management address and receive replies
2. **Step Two**
   Create a **`ClientMessage`**
3. **Step Three**
   Use the helper class **`org.hornetq.api.core.management.ManagementHelper`** to fill the message with the management properties.
4. **Step Four**
   Send the message using the **`ClientRequestor`**
5. **Step Five**
   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`**, and so on).

28.3.1. Configuring Core Management

The management address to send management messages is configured in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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>
```

28.4. 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 as 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");
```

28.4.1. Configuring JMS Management

Whether JMS or the core API is used for management, the configuration steps are the same (see Section 28.3.1, "Configuring Core Management").

28.5. MANAGEMENT NOTIFICATIONS

HornetQ emits *notifications* to inform listeners of potentially interesting events (creation of new resources, security violation, and so on).

These notifications can be received three different ways:

- JMX notifications
- Core messages
- JMS messages

28.5.1. JMX Notifications

If JMX is enabled (see Section 28.2.1, "Configuring JMX"), JMX notifications can be received by subscribing to two MBeans:

- **`org.hornetq:module=Core,type=Server`** for notifications on *core* resources
- **`org.hornetq:module=JMS,type=Server`** for notifications on *JMS* resources

28.5.2. Core Messages Notifications

HornetQ defines a *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.

Notification messages are regular core messages with additional properties corresponding to the notification (its type, when it occurred, the resources which were concerned, and so on).

Since notifications are regular core messages, it is possible to use message selectors to filter out notifications and receive only a subset of all the notifications emitted by the server.

28.5.2.1. Configuring The Core Management Notification Address

The management notification address to receive management notifications is configured in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**:

```
<management-notification-address>hornetq.notifications</management-
notification-address>
```

By default, the address is **hornetq.notifications**.

28.5.3. JMS Messages Notifications

HornetQ 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).

You must change the server's management notification address to start with **jms.queue** (if it is a JMS queue) or **jms.topic** (if it is a JMS topic) to use a JMS Destination to receive management notifications.

```
<!-- notifications will be consumed from "notificationsTopic" JMS Topic --
>
<management-notification-address>jms.topic.notificationsTopic</management-
notification-address>
```

Once the notification topic is created, you can receive messages from it or set a **MessageListener**:

```
   Topic notificationsTopic =
HornetQJMSClient.createTopic("notificationsTopic");

   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("------------------------");
        }
    });
```

28.6. MESSAGE COUNTERS

Message counters can be used to obtain information on queues over time as HornetQ keeps a history of 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 intervals. You could also view this information using the JMX Console, or use the core API (**`org.hornetq.api.core.management.MessageCounterInfo`**) to extract the information.

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 as messages consumed from it. The number of messages in the queue remains the same in both cases but its use is different.

Message counters provide 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 or 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.

`lastAddTimestamp`

The time stamp of the last time a message was added to the queue.

`udpateTimestamp`

The time stamp of the last message counter update.

28.6.1. Configuring Message Counters

Message counters are disabled by default as they could have a negative effect on memory.

To enable message counters, you can set it to **`true`** in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**:

```
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keep a history of the queue metrics (10 days by default) and sample all the queues at regular intervals (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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());
```

28.7. ADMINISTERING HORNETQ RESOURCES USING THE ADMIN CONSOLE

It is possible to create and configure HornetQ resources via the admin console within the JBoss Enterprise Application Platform.

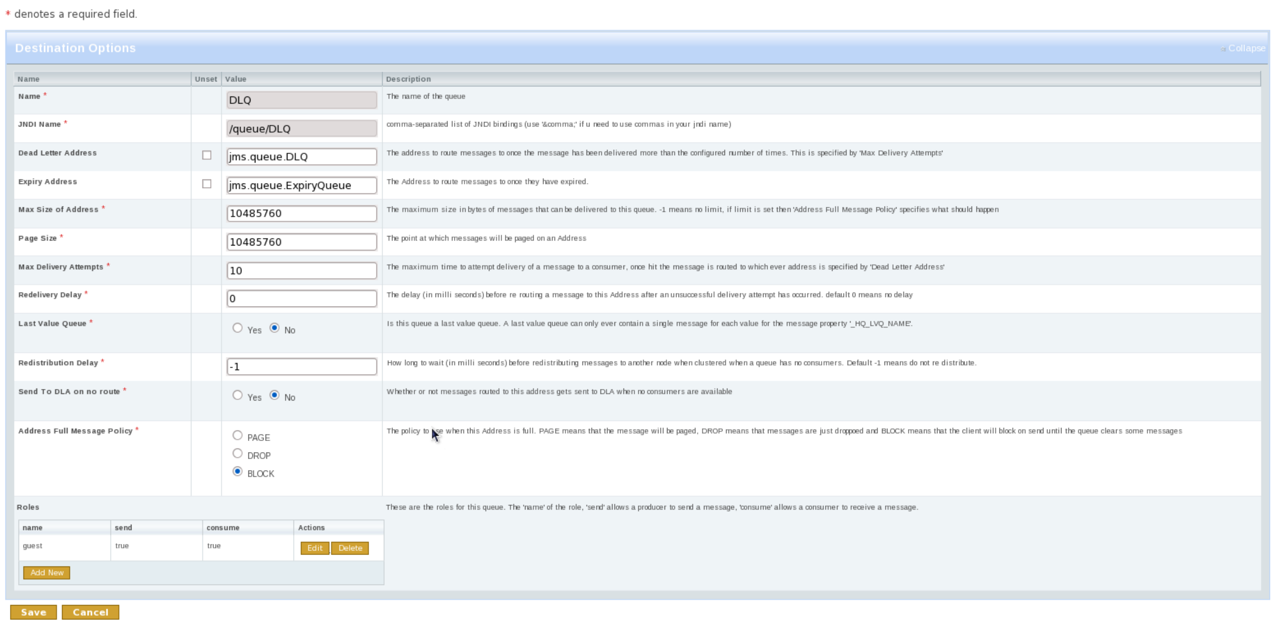
The admin console will allow you to create destinations (JMS topics and queues) and JMS connection factories.

Once logged in to the admin console you will see a JMS Manager item in the left hand tree. All HornetQ resources will be configured via this. This will have child items for JMS queues, topics and connection factories. Clicking on each node will reveal which resources are currently available. The following sections explain how to create and configure each resource in turn.

28.7.1. JMS Queues

To create a new JMS queue click on the JMS Queues item to reveal the available queues. On the right hand panel you will see an **Add a New Resource** button. Click on this and then choose the default (JMS Queue) template. Click **Continue**. The important things to fill in here are the name of the queue and the JNDI name of the queue. The JNDI name is what you will use to look up the queue in JNDI from your client. For most queues this will be the only info you will need to provide, as sensible defaults are provided for the others. You will also see a security roles section near the bottom. If you do not provide any roles for this queue then the server's default security configuration will be used. After you have created the queue these will be shown in the configuration. All configuration values, except the name and JNDI name, can be changed via the configuration tab after clicking on the queue in the admin console. The following section explains these in more detail.

After highlighting the configuration you will see the following screen:



* denotes a required field.

Name	Unset	Value	Description
Name *		DLQ	The name of the queue
JNDI Name *		/queue/DLQ	comma-separated list of JNDI bindings (use '&comma;' if u need to use commas in your jndi name)
Dead Letter Address	☐	jms.queue.DLQ	The address to route messages to once the message has been delivered more than the configured number of times. This is specified by 'Max Delivery Attempts'
Expiry Address	☐	jms.queue.ExpiryQueue	The Address to route messages to once they have expired.
Max Size of Address *		10485760	The maximum size in bytes of messages that can be delivered to this queue. -1 means no limit, if limit is set then 'Address Full Message Policy' specifies what should happen
Page Size *		10485760	The point at which messages will be paged on an Address
Max Delivery Attempts *		10	The maximum time to attempt delivery of a message to a consumer, once hit the message is routed to which ever address is specified by 'Dead Letter Address'
Redelivery Delay *		0	The delay (in milli seconds) before re routing a message to this Address after an unsuccessful delivery attempt has occurred. default 0 means no delay
Last Value Queue *		Yes [✓ No]	Is this queue a last value queue. A last value queue can only ever contain a single message for each value for the message property '_HQ_LVQ_NAME'.
Redistribution Delay *		-1	How long to wait (in milli seconds) before redistributing messages to another node when clustered when a queue has no consumers. Default -1 means do not re distribute.
Send To DLA on no route *		Yes [✓ No]	Whether or not messages routed to this address gets sent to DLA when no consumers are available
Address Full Message Policy *		PAGE DROP [✓ BLOCK]	The policy to use when this Address is full. PAGE means that the message will be paged, DROP means that messages are just droppoed and BLOCK means that the client will block on send until the queue clears some messages
Roles			These are the roles for this queue. The 'name' of the role, 'send' allows a producer to send a message, 'consume' allows a consumer to receive a message.

name	send	consume	Actions
guest	true	true	Edit Delete

Add New

The name and JNDI name cannot be changed. Recreate the queue with the appropriate settings if you want to change these names. The rest of the configuration options, apart from security roles, relate to address settings for a particular address. The default address settings are picked up from the server configuration. If you change any of these settings or create a queue via the console, a new Address Settings entry will be added. For a full explanation on Address Settings, see Section 23.3, “Configuring Queues Through Address Settings”

To delete a queue, click on the **Delete** button beside the queue name in the main JMS queues screen. This will also delete any address settings or security settings previously created for the queue address.

The last part of the configuration options are security roles. If none are provided on creation then the server's default security settings will be shown. If these are changed or updated, new security settings are created for the address of this queue. For more information on security settings, see Chapter 29, *Security*.

It is also possible to view statistics for this queue via the metrics tab. This will show statistics such as message count, consumer count, and so on.

Operations can be performed on a queue via the control tab. This will allow you to start and stop the queue, list, move, expire, and delete messages from the queue and other useful operations. To invoke an operation, click on the button for the operation you want. This will take you to a screen where parameters for the operation can be set. Once set, click the **OK** button to invoke the operation. Results will appear at the bottom of the screen.

28.7.2. JMS Topics

Creating and configuring JMS topics is almost identical to creating queues. The only difference is that the configuration will be applied to the queue representing a subscription.

28.7.3. JMS Connection Factories

The format for creating connection factories is similar to JMS queues and topics. For a list of all the connection factory settings, refer to Section A.1.4, “hornetq-jms.xml”.

CHAPTER 29. SECURITY

This chapter explores how security works with HornetQ and how it can be configured.

For performance reasons, security is cached and invalidated periodically. To change this period, set the property **`security-invalidation-interval`**, which is in milliseconds. The default is **`10000`** ms.

WARNING

Security is enabled by default, to ensure your production system security remains high once properly configured. You can disable security completely by explicitly setting the **`security-enabled`** property to **`false`** in the **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** file. This practice is strongly discouraged for production environments.

29.1. 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 7, *Using Core*, HornetQ core primarily consists 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 sets 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.

`createNonDurableQueue`

This permission allows the user to create a non-durable queue under matching addresses.

`deleteNonDurableQueue`

This permission allows the user to delete a non-durable 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 is specified of the roles that are granted that permission. If the user has any of those roles, they will be granted that permission for that set of addresses.

An example security block is described in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**:

```
<security-setting match="globalqueues.europe.#">
   <permission type="createDurableQueue" roles="admin"/>
   <permission type="deleteDurableQueue" roles="admin"/>
   <permission type="createNonDurableQueue" roles="admin, guest, europe-
users"/>
   <permission type="deleteNonDurableQueue" 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 refer to Chapter 11, *Understanding the HornetQ Wildcard Syntax*. 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."

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 Enterprise Application Platform security.

For more information on configuring the security manager, refer to Section 29.4, "Changing the security manager".

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.

```
<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`**, **`createNonDurableQueue`**, **`deleteNonDurableQueue`** are not inherited from the other security-setting block.

By not inheriting permissions, it allows effective denial of permissions in more specific security-setting blocks by not specifying them. Otherwise, it would not be possible to deny permissions in sub-groups of addresses.

29.2. SECURE SOCKETS LAYER (SSL) TRANSPORT

When messaging clients are connected to servers, or servers are connected to other servers (for example, via bridges) over an untrusted network, HornetQ allows that traffic to be encrypted using the Secure Sockets Layer (SSL) transport.

For more information on configuring the SSL transport, refer to Chapter 14, *Configuring the Transport*.

29.3. BASIC USER CREDENTIALS

HornetQ ships with a security manager implementation that reads user credentials (user names and passwords), and role information from the **`hornetq-users.properties`** and **`hornetq-users.roles`** files. These files are both located in the **`/conf/props/`** directory within the profile you wish to run.

User credentials, and roles, can easily be added into these files.

Example 29.1, “hornetq-users.properties example file” and Example 29.2, “hornetq-users.roles example file” contain four users. Each user is specified in both the .properties and .roles files.

Following the syntax in each file's comments, you assign each user a unique password and attach roles to each user to control what parts of HornetQ they can change.

Example 29.1. hornetq-users.properties example file

```
#
# user=password
#
guest=guest
tim=marmite
andy=doner_kebab
jeff=camembert
```

Example 29.2. hornetq-users.roles example file

```
#
# user=role1,role2,...
#
guest=guest
tim=admin
andy=admin,guest
jeff=europe-users,guest
```

The first thing to note is the guest user defined in both files. A user is classed as a guest when the client does not specify a user name/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: tim, who has the role *admin*; andy, who has the roles *admin* and *guest*; and jeff, who has the roles *europe-users* and *guest*.

29.4. CHANGING THE SECURITY MANAGER

If you do not want to use the default security manager, you can specify a different one by editing the file **`hornetq-jboss-beans.xml`** and changing the class for the **`HornetQSecurityManager`** bean.

HornetQ ships with two security manager implementations you can use. One is a JAAS security manager and the other is for integrating with JBoss Enterprise Application Platform security. Alternatively you could write your own implementation by implementing the **`org.hornetq.spi.core.security.HornetQSecurityManager`** interface, and specifying the class name of your implementation in the file **`hornetq-jboss-beans.xml`**.

These two implementations are discussed in the next two sections.

29.5. JAAS SECURITY MANAGER

JAAS (Java Authentication and Authorization Service) is a standard part of the Java platform. It provides a common API for security authentication and authorization, allowing you to plug in 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 is an example:

```
<bean name="HornetQSecurityManager"
      class="org.hornetq.integration.jboss.security.JAASSecurityManager">
    <start ignored="true"/>
    <stop ignored="true"/>

    <property
name="ConfigurationName">org.hornetq.jms.example.ExampleLoginModule</prope
rty>
    <property name="Configuration">
       <inject bean="ExampleConfiguration"/>
    </property>
    <property name="CallbackHandler">
       <inject bean="ExampleCallbackHandler"/>
    </property>
</bean>
```

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

29.6. HORNETQ SECURITY MANAGER

The HornetQ security manager provides tight integration with the JBoss Enterprise Application Platform security model.

The class name of this security manager is **`org.hornetq.integration.jboss.security.JBossASSecurityManager`**

An example of how the **`JBossASSecurityManager`** is configured is described in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-jboss-beans.xml`**.

29.6.1. Configuring Client Login

JBoss Enterprise Application Platform can be configured to allow client login. This is when a Java EE component such as a Servlet or EJB sets security credentials on the current security context, and these are used throughout the call.

HornetQ can use these settings when sending or consuming messages by changing the allowClientLogin property to **`true`** (default is **`false`**) in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-jboss-beans.xml`** file. This bypasses HornetQ authentication, and propagates the provided Security Context.

If HornetQ should authenticate using the propagated security, set the **`authoriseOnClientLogin`** to true in addition to ***`allowClientLogin`***.

29.7. CHANGING THE SECURITY DOMAIN

To change the security domain, add a ***`securityDomainName`*** property to the **`HornetQSecurityManager`** bean in **`hornetq-jboss-beans.xml`**.

The **`HornetQSecurityManager`** bean does not contain this property by default.

Example 29.3. `HornetQSecurityManager` bean

```
<!-- The security manager -->
<bean name="HornetQSecurityManager"
class="org.hornetq.integration.jboss.security.JBossASSecurityManager">
   <start ignored="true"/>
   <stop ignored="true"/>
   <depends>JBossSecurityJNDIContextEstablishment</depends>
   <property name="allowClientLogin">false</property>
   <property name="authoriseOnClientLogin">false</property>
   <property name="securityDomainName">java:/jaas/hornetq</property>
 1

</bean>
```

1	The example above shows the ***securityDomainName*** property as it should be formatted, if used. Note that the security domain shown in this example is the system default and will be used unless the ***securityDomainName*** parameter has been added with a different value.

29.8. CHANGING THE USER NAME/PASSWORD FOR CLUSTERING

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The user name/password they use for this should always be changed from the installation default to prevent a security risk.

Refer to Chapter 28, *Management* for instructions on how to do this.

CHAPTER 30. APPLICATION SERVER INTEGRATION AND JAVA EE

Since HornetQ also provides a JCA adapter, it is also possible to integrate HornetQ as a JMS provider in other Java EE 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 (MDBs) and the outflow of messages sent from other Java EE components, (for example, EJBs and Servlets).

This section explains the basics behind configuring the different Java EE components in the AS.

30.1. CONFIGURING MESSAGE-DRIVEN BEANS

Message delivery to an MDB using HornetQ is configured on the JCA Adapter in the**<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/jms-ra.rar/META-INF/ra.xml** file. By default this is configured to consume messages using an InVM connector from the instance of HornetQ running within the application server. If you need to adjust the configuration parameters, parameter details can be found in Section 30.4, “Configuring the JCA Adaptor”.

HornetQ provides standard configuration in the default installation so MDBs can reference the Resource Adapter destination and destination type.

The **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/jms-ds.xml** data source file links destination and destination type configuration information in the **ra.xml** file using the <rar-name> directive.

30.1.1. 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, it will try to 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 =
"javax.jms.Queue"),
    @ActivationConfigProperty
      (propertyName = "destination", propertyValue = "queue/testQueue")
  })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
public class MDB_CMP_TxRequiredExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

The **TransactionManagement** annotation tells the container to manage the transaction. 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
   MessageDrivenContextContext ctx;

   public void onMessage(Message message)
   {
      try
      {
         //something here fails
      }
      catch (Exception e)
      {
         ctx.setRollbackOnly();
      }
   }
```

If you do not want the overhead of an XA transaction being created every time but you would still like the message delivered within a transaction (for example, 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 =
"javax.jms.Queue"),
    @ActivationConfigProperty
      (propertyName = "destination", propertyValue = "queue/testQueue"),
    @ActivationConfigProperty
      (propertyName = "useLocalTx", propertyValue = "true")
  })
@TransactionManagement(value = TransactionManagementType.CONTAINER)
@TransactionAttribute(value = TransactionAttributeType.NOT_SUPPORTED)
public class MDB_CMP_TxLocalExample implements MessageListener
{
   public void onMessage(Message message)...
}
```

30.1.2. 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 =
"javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "acknowledgeMode", propertyValue = "Dups-ok-
acknowledge")
    })
@TransactionManagement(value= TransactionManagementType.BEAN)
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 life cycle 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();
    }
  }
```

30.1.3. Using Message Selectors with Message-Driven Beans

It is also possible to use MDBs with message selectors. To do this simple define your message selector as follows:

```
@MessageDriven(name = "MDBMessageSelectorExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "messageSelector", propertyValue = "color =
'RED'")
```

```
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
public class MDBMessageSelectorExample implements MessageListener
{
   public void onMessage(Message message)....
}
```

30.1.4. High Availability in Message-driven Beans

For message-driven beans to be compatible with High Availability (HA) environments, you must set an @ActivationConfigProperty relating to HA in the bean.

IMPORTANT

Not all server profiles are enabled for clustering by default. Ensure the <clustered>true</clustered> directive is set in ***<JBOSS_HOME>*/jboss-as/server/*<PROFILE>*/deploy/hornetq/hornetq-configuration.xml**. The Activation Property will cause a **HornetQException** if the <clustered> directive is not correctly specified. For more information about Clustering, refer to Chapter 36, *Clusters*.

Add an activation property to the bean's **activationConfig** block to make the MDB compatible with HA environments:

```
activationConfig =
    {
     @ActivationConfigProperty
        (propertyName = "hA", propertyValue = "true"),
    }
```

For more information about High Availability, refer to Chapter 37, *High Availability and Fail-over*

30.2. SENDING MESSAGES FROM WITHIN JAVA EE COMPONENTS

The JCA adapter can also be used for sending messages. The Connection Factory to use is configured by default in the ***<JBOSS_HOME>*/jboss-as/server/*<PROFILE>*/deploy/hornetq/jms-ds.xml** file and is mapped to **java:/JmsXA**.

Using this from within a Java EE component specifies that message sending is done as part of the JTA transaction being used by the component. If message sending fails, the overall transaction is rolled back and the message is re-sent. Here is an example of this from within an MDB:

```
@MessageDriven(name = "MDBMessageSendTxExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue")
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
```

```
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)
            {
            }
         }
      }
   }
   }
```

You can also use the JMS JCA adapter for sending messages from EJBs (including Session, Entity and Message-Driven Beans), Servlets (including JSPs), and custom MBeans.

30.3. MDB AND CONSUMER POOL SIZE

Most application servers, including JBoss, allow you to configure how many MDBs there are in a pool. It is important to understand that the ***MaxPoolSize*** parameter in the **ejb3-interceptors-aop.xml** file will *not* have an effect on how many sessions or consumers are created because the Resource Adaptor implementation is not aware of the application server MDB implementation.

For example, if you set the MDB ***MaxPoolSize*** to 1, 15 sessions or consumers are created (15 is the default). To limit how many sessions or consumers are created, set the **maxSession** parameter on the resource adapter, or through an ***ActivationConfigProperty*** annotation on the MDB.

```
@MessageDriven(name = "MDBMessageSendTxExample",
  activationConfig =
    {
      @ActivationConfigProperty
        (propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
      @ActivationConfigProperty
        (propertyName = "destination", propertyValue = "queue/testQueue"),
      @ActivationConfigProperty
        (propertyName = "maxSession", propertyValue = "1")
    })
@TransactionManagement(value= TransactionManagementType.CONTAINER)
@TransactionAttribute(value= TransactionAttributeType.REQUIRED)
public class MyMDB implements MessageListener
{ ....}
```

30.4. CONFIGURING THE JCA ADAPTOR

The Java Connector Architecture (JCA) Adapter is what allows HornetQ to be integrated with Java EE components such as MDBs and EJBs. It configures how components such as MDBs consume messages from the HornetQ server and also how components such as EJBs or Servlets can send messages.

The HornetQ JCA adapter is deployed via the **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/jms-ra.rar** archive. The configuration of the adapter is found in this archive under **META-INF/ra.xml**.

Because of the size of this file, it is included in Appendix B, *ra.xml HornetQ Resource Adapter File* if you need to reference it. You can also open this file from the specified directory and review it in conjunction with the following information.

There are three main parts to this configuration, which are described in the following sections.

1. Section 30.4.1, “JCA Global Properties”
2. Section 30.4.2, “JCA Outbound Configuration”
3. Section 30.4.3, “JCA Inbound Configuration”

30.4.1. JCA 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 two properties configure the transport used by the adapter and the rest configure the connection factory itself.

NOTE

All connection factory properties will use the defaults if they are not provided, except for the **`reconnectAttempts`** which will default to -1. This signifies that the connection should attempt to reconnect on connection failure indefinitely. 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 30.1. Global Configuration Properties

Property Name	Property Type	Property Description
`ConnectorClassName`	String	The Connector class name (see Chapter 14, *Configuring the Transport* for more information)
`ConnectionParameters`	String	The transport configuration. These parameters must be in the form of **`key1=val1;key2=val2;`** and will be specific to the connector used
`useLocalTx`	boolean	True will enable local transaction optimization.
`UserName`	String	The user name to use when making a connection
`Password`	String	The password to use when making a connection
`BackupConnectorClassName`	String	The backup transport to use in case of failure of the live node
`BackupConnectionParameters`	String	The backup transport configuration parameters
`DiscoveryAddress`	String	The discovery group address to use to auto-detect a server
`DiscoveryPort`	Integer	The port to use for discovery
`DiscoveryRefreshTimeout`	Long	The timeout, in milliseconds, to refresh.

Property Name	Property Type	Property Description
DiscoveryInitialWaitTimeout	Long	The initial time to wait for discovery.
ConnectionLoadBalancingPolicyClassName	String	The load balancing policy class to use.
ConnectionTTL	Long	The time to live (in milliseconds) for the connection.
CallTimeout	Long	the call timeout (in milliseconds) for each packet sent.
DupsOKBatchSize	Integer	the batch size (in bytes) between acknowledgments when using DUPS_OK_ACKNOWLEDGE mode
TransactionBatchSize	Integer	the batch size (in bytes) between acknowledgments when using a transactional session
ConsumerWindowSize	Integer	the window size (in bytes) for consumer flow control
ConsumerMaxRate	Integer	the fastest rate a consumer may consume messages per second
ConfirmationWindowSize	Integer	the window size (in bytes) for reattachment confirmations
ProducerMaxRate	Integer	the maximum rate of messages per second that can be sent
MinLargeMessageSize	Integer	the size (in bytes) before a message is treated as large
BlockOnAcknowledge	Boolean	whether or not messages are acknowledged synchronously
BlockOnNonDurableSend	Boolean	whether or not non-durable messages are sent synchronously
BlockOnDurableSend	Boolean	whether or not durable messages are sent synchronously
AutoGroup	Boolean	whether or not message grouping is automatically used

Property Name	Property Type	Property Description
`PreAcknowledge`	Boolean	whether messages are pre acknowledged by the server before sending
`ReconnectAttempts`	Integer	maximum number of retry attempts, default for the resource adapter is -1 (infinite attempts)
`RetryInterval`	Long	the time (in milliseconds) to retry a connection after failing
`RetryIntervalMultiplier`	Double	multiplier to apply to successive retry intervals
`FailoverOnServerShutdown`	Boolean	If true client will reconnect to another server if available
`ClientID`	String	the pre-configured client ID for the connection factory
`ClientFailureCheckPeriod`	Long	the period (in ms) after which the client will consider the connection failed after not receiving packets from the server
`UseGlobalPools`	Boolean	whether or not to use a global thread pool for threads
`ScheduledThreadPoolMaxSize`	Integer	the size of the *scheduled thread* pool
`ThreadPoolMaxSize`	Integer	the size of the thread pool
`SetupAttempts`	Integer	Number of attempts to setup a JMS connection (default is 10, -1 means to attempt infinitely). It is possible that the MDB is deployed before the JMS resources are available. In that case, the resource adapter will try to setup several times until the resources are available. This applies only for inbound connections
`SetupInterval`	Long	Interval in milliseconds between consecutive attempts to setup a JMS connection (default is 2000m). This applies only for inbound connections

30.4.2. JCA Outbound Configuration

The outbound configuration remains unchanged because it defines 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`**. A default **`jms-ds.xml`** configuration file is located in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/jms-ds.xml`**. The connection factories defined in this file inherit their properties from the main **`ra.xml`** configuration but can also be overridden. The following example shows how to override them.

```
<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.core.remoting.impl.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 is 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 overrides those in the **`ra.xml`** configuration file. Any of the elements pertaining to the connection factory can be overridden here.

The outbound configuration also defines additional properties in addition to the global configuration properties.

Table 30.2. Outbound Configuration Properties

Property Name	Property Type	Property Description
SessionDefaultType	String	the default session type
UseTryLock	Integer	try to obtain a lock within specified number of seconds. less than or equal to 0 disable this functionality

30.4.3. JCA Inbound Configuration

The inbound configuration should again remain unchanged. This controls what forwards messages onto MDBs. 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 inbound configuration also defines additional properties in addition to the global configuration properties.

Table 30.3. Inbound Configuration Properties

Property Name	Property Type	Property Description
Destination	String	JNDI name of the destination
DestinationType	String	type of the destination, either `javax.jms.Queue` or `javax.jms.Topic` (default is javax.jms.Queue)
AcknowledgeMode	String	The Acknowledgment mode, either `Auto-acknowledge` or `Dups-ok-acknowledge` (default is Auto-acknowledge). `AUTO_ACKNOWLEDGE` and `DUPS_OK_ACKNOWLEDGE` are acceptable values.
MaxSession	Integer	Maximum number of session created by this inbound configuration (default is 15)
MessageSelector	String	the message selector of the consumer
SubscriptionDurability	String	Type of the subscription, either `Durable` or `NonDurable`
SubscriptionName	String	Name of the subscription
TransactionTimeout	Long	The transaction timeout in milliseconds (default is 0, the transaction does not timeout)
UseJNDI	Boolean	Whether or not use JNDI to look up the destination (default is true)

30.4.4. 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 Clustering section of the *Administration and Configuration Guide* for this release of JBoss Enterprise Application Platform.

30.4.5. XA Recovery

`XA recovery` deals with system or application failures to ensure that resources 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.

XA Recovery is pre-configured in HornetQ from version 2.2.8.GA. No configuration is required and the service cannot be disabled.

HornetQ takes 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.

For more information on XA Recovery, refer to the JBoss Transactions documentation provided with this release of JBoss Enterprise Application Platform.

CHAPTER 31. THE JMS BRIDGE

HornetQ includes a fully-functional JMS message bridge, which consumes messages from a source queue or topic and sends them to a target queue or topic, usually 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.

The bridge can also be used to bridge messages from other non-HornetQ JMS servers that are JMS 1.1 compliant.

IMPORTANT

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 34, *Core Bridges*) is used to bridge any two HornetQ instances and uses the core API. A core bridge will typically provide better performance than a JMS bridge, and provides 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 (for example, due to network failure), the bridge will attempt to reconnect to the target server until it comes back on line, at which point operations will resume 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.

The bridge is typically deployed by JBoss Microcontainer using a beans configuration file (**`jms-bridge-jboss-beans.xml`**), which is located in the **`JBOSS_DIST/jboss-as/server/PROFILE/deploy`** directory.

Example 31.1, "jms-bridge-jboss-beans.xml Sample Config" shows a configuration sample that bridges two destinations on the same server.

Example 31.1. jms-bridge-jboss-beans.xml Sample Config

```
<?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 user name here) -->
         <parameter><null /></parameter>
         <!-- Source Password (no password here)-->
         <parameter><null /></parameter>
         <!-- Target User Name (no user name 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 resources         -->
  <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>
```

31.1. JMS BRIDGE PARAMETERS

The **`JMSBridge`** bean, as shown in Example 31.1, “jms-bridge-jboss-beans.xml Sample Config”, is configured via parameters passed to its constructor in a particular order. This order, and a description of each parameter, is outlined in the list following.

NOTE

To leave a parameter 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`

Injects the **`SourceCFF`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the **`ConnectionFactory`**.

`Target Connection Factory Factory`

Injects the **`TargetCFF`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the target **`ConnectionFactory`**.

`Source Destination Factory Factory`

Injects the **`SourceDestinationFactory`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the source **`Destination`**.

`Target Destination Factory Factory`

Injects the **`TargetDestinationFactory`** bean defined in the **`jms-bridge-jboss-beans.xml`** file, which creates the target **`Destination`**.

`Source User Name`

Defines the username used to create the source connection.

`Source Password`

Defines the password for the user name used to create the source connection.

`Target User Name`

Defines the user name used to create the target connection.

`Target Password`

Defines the password of the user name used to create the target connection.

`Selector`

Specifies a JMS selector expression used when 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.

`Failure Retry Interval`

Specifies the time in milliseconds to wait in between attempting to recreate connections to the source or target servers when the bridge detects a connection failure.

`Max Retries`

Specifies the number of times to attempt to recreate connections to the source or target servers when the bridge detects a connection failure. The bridge will stop trying to reconnect after this number of tries. `-1` means 'try forever'.

`Quality of Service`

Defines the quality of service mode. The possible values are:

- `AT_MOST_ONCE`
- `DUPLICATES_OK`
- `ONCE_AND_ONLY_ONCE`

See Section 31.4, "Quality Of Service Modes" for a explanation of these modes.

`Max Batch Size`

Defines the maximum number of messages that should be consumed from the source connection before the messages are sent in a batch to the target destination. Its value must be `1` or greater.

`Max Batch Time`

Defines the number of milliseconds to wait before sending a batch to the target destination, even if the number of messages consumed has not reached `MaxBatchSize`. Its value must be `1` or greater, or `-1` to specify 'wait forever'.

`Subscription Name`

If the source destination is a topic, and you want to consume from the topic with a durable subscription, this parameter defines the durable subscription name.

`Client ID`

If the source destination is a topic, and you want to consume from the topic with a durable subscription, this parameter defines the JMS client ID to use when creating or looking up the durable subscription.

`Add MessageID In Header`

When **`true`**, the original message's message ID is appended to the message sent to the destination in the **`HORNETQ_BRIDGE_MSG_ID_LIST`** header. If the message is bridged multiple times, each message ID is appended. This lets you use a distributed response pattern.

NOTE

When a message is received, a response can be sent using the correlation ID of the first message ID so that when the original sender receives the response it is able to correlate the message.

`MBean Server`

Set this to the place where the JMS Bridge is registered (the application server MBeanServer) to manage the JMS Bridge with JMX.

`ObjectName`

If **`MBeanServer`** is set, this parameter must be set to define the name used to register the JMS Bridge MBean. This name must be unique.

31.2. SOURCE AND TARGET CONNECTION FACTORIES

The source and target connection factory factories are used to create the connection factory that creates the connection for the source or target server.

Example 31.1, "jms-bridge-jboss-beans.xml Sample Config" uses the default implementation provided by HornetQ, which looks up the connection factory using JNDI. For other JMS providers, a different implementation may be required. To do so, implement the **`org.hornetq.jms.bridge.ConnectionFactoryFactory`** interface.

31.3. SOURCE AND TARGET DESTINATION FACTORIES

These create or look up the destinations.

Example 31.1, "jms-bridge-jboss-beans.xml Sample Config" uses the default implementation provided by HornetQ that looks up the destination using JNDI.

Different implementations can be provided with the **`org.hornetq.jms.bridge.DestinationFactory`** interface.

31.4. QUALITY OF SERVICE MODES

The quality of service modes used by the bridge are described here in more detail.

31.4.1. AT_MOST_ONCE

Messages reach the destination once at most. Messages are consumed from the source and acknowledged before they are sent to the destination. If failure occurs between leaving the source and arriving at the destination, messages can be lost. This mode is available for both durable and non-durable messages.

31.4.2. DUPLICATES_OK

Messages are consumed from the source and acknowledged after they are successfully sent to the destination. If failure occurs after messages are sent, but before acknowledgment is returned, messages can be sent again once the system recovers, so the destination may receive duplicates after a failure. This mode is available for both durable and non-durable messages.

31.4.3. ONCE_AND_ONLY_ONCE

Messages reach the destination exactly once. If both source and destination are on the same HornetQ server instance, this mode sends and acknowledges messages as part of the same local transaction. If the source and destination are on different servers, this mode enlists the sending and consumption sessions in a JTA transaction.

This JTA transaction is controlled by JBoss Transactions JTA, a full-recovery transaction manager which provides a very high degree of durability. If JTA is required, both supplied connection factories must be **`XAConnectionFactory`** implementations.

This is likely to be the slowest mode, since it requires extra persistence. This mode is only available for durable messages.

NOTE

For some applications, once and only once semantics could be provided by setting the **`DUPLICATES_OK`** mode and then checking for and discarding duplicate messages on the destination. This approach is not as reliable as using **`ONCE_AND_ONLY_ONCE`** mode, but may be a useful alternative.

31.4.4. Time outs and the JMS bridge

There is a possibility that the target or source server will not be available at some point in time. If this occurs then the bridge will try **`Max Retries`** to reconnect every **`Failure Retry Interval`** milliseconds as specified in the JMS Bridge definition.

However since a third party JNDI is used, in this case the JBoss naming server, it is possible for the JNDI lookup to hang if the network were to disappear during the JNDI lookup. To stop this occurring the JNDI definition can be configured to time out if this occurs. To do this set the **`jnp.timeout`** and the **`jnp.sotimeout`** on the Initial Context definition. The first sets the connection timeout for the initial connection and the second the read timeout for the socket.

NOTE

Once the initial JNDI connection has succeeded all calls are made using Remote Method Invocation (RMI). If you want to control the timeouts for the RMI connections then this can be done via system properties. JBoss uses Oracle's RMI and the properties can be found here. The default connection timeout is 10 seconds and the default read timeout is 18 seconds.

If you implement your own factories for looking up JMS resources then you will have to bear in mind timeout issues.

CHAPTER 32. CLIENT RECONNECTION AND SESSION REATTACHMENT

HornetQ clients can be configured to automatically reconnect or re-attach to the server if a failure is detected in the server-client connection.

32.1. 100% TRANSPARENT SESSION RE-ATTACHMENT

If failure is transient (that is, due to a temporary network failure) and the target server was not restarted, sessions will still exist on the server as long as the ***`connection-ttl`*** has not expired (see Chapter 15, *Detecting Dead Connections* for details about ***`connection-ttl`***).

In this case, HornetQ automatically re-attaches the client sessions to the server sessions when the connection reconnects. This is done 100% transparently and the client can continue as before.

As HornetQ clients send commands to their servers, they store each sent command in an in-memory buffer. When connection failure occurs and the client re-attaches to its server, the server passes the ID of the last command it successfully received back to the client as part of the re-attachment protocol. If the client had attempted to send any other commands, it can resend those commands from its buffer so that client and server can reconcile their differences.

The **`ConfirmationWindowSize`** parameter (typically set in the **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/jms-ra.rar/META-INF/ra.xml`** file) defines the size of the buffer in bytes. When the server has received **`ConfirmationWindowSize`** bytes of commands and processed them it will send back a command confirmation to the client. The client can then remove confirmed commands from the buffer.

Setting **`ConfirmationWindowSize`** to **`-1`** (default) disables buffering and prevents re-attachment from occurring, forcing reconnect instead.

If you are using JMS, and the JMS service on the server is being used to load your JMS connection factory instances into JNDI, then uncomment the **`ConfirmationWindowSize`** parameter in the **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/jms-ra.rar/META-INF/ra.xml`** file. If you are using JMS, but not JNDI, set these values directly on the **`HornetQConnectionFactory`** instance with the appropriate setter method. If you are using Core, you can set these values directly on the **`ClientSessionFactory`** instance with the appropriate setter method.

32.2. SESSION RECONNECTION

If the server is restarted after crashing or being stopped, sessions will no longer exist on the server and it will not be possible to re-attach completely transparently.

In this case, HornetQ automatically reconnects the connection and recreates any sessions and consumers on the server based on the sessions and consumers of the client. (This process is identical to fail-over onto a backup server.) Client reconnection is also used internally by components such as core bridges to let them reconnect to their target servers. See Section 37.2.1, “Automatic Client fail-over” for a full explanation of how transacted and non-transacted sessions are reconnected during fail-over, and the requirements of once-and-only-once delivery guarantees.

32.3. CONFIGURING RECONNECTION/REATTACHMENT ATTRIBUTES

If you are using JMS and the JMS Service on the server is being used to load your JMS connection factory instances into JNDI, you can specify these parameters in ***JBOSS_DIST*/jboss-as/server/*<PROFILE>*/deploy/hornetq/hornetq-jms.xml** like so:

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

If you are using JMS, but instantiating your JMS connection factory directly, you can specify the parameters using the appropriate setter methods on the **HornetQConnectionFactory** immediately after its creation.

If you are 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 its creation.

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 will be unable to re-attach, and any **ExceptionListener** or **SessionFailureListener** instances registered on the connection or session will be called.

NOTE

Registered JMS **ExceptionListener** or Core **SessionFailureListener** instances are called when a client reconnects or re-attaches.

CHAPTER 33. DIVERTING AND SPLITTING MESSAGE FLOWS

Diverts are objects that transparently divert messages routed to one address to some other address, without making any changes to any client application logic.

Diverts can be *exclusive* (messages are diverted to the new address and do not go to the previous address at all), or *non-exclusive* (a copy of the message is sent to the new address, and the original message goes to the old address). Non-exclusive diverts can therefore be used for *splitting* message flows, for example, when every order sent to an order queue must be monitored.

Diverts can also be configured with a filter, so that only messages that match the filter are diverted.

Diverts can also be configured to apply a **`Transformer`**. If specified, all diverted messages can be transformed by this **`Transformer`**.

Diverts only move messages to addresses on the same server, but when used in combination with bridges more complex routings can be set up. One common use case to "divert" to a different server is to divert messages to a local store-and-forward queue and then set up a bridge to consume from that queue and forward consumed messages to an address on another server. The diverts on a server can be thought of as a routing table for messages. Combining diverts with bridges allow you to create a distributed network of reliable routing connections between multiple geographically distributed servers.

Diverts are defined in the **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** file. There can be zero or more diverts in the file.

33.1. EXCLUSIVE DIVERTS

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.

Diverts are defined in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** using the following directives:

```
<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>
```

The **`prices-divert`** divert specified above diverts any messages sent to the address **`jms.topic.priceUpdates`** (a local JMS Topic called **`priceUpdates`**) to another local address **`jms.queue.priceForwarding`** (a local JMS Queue called **`priceForwarding`**).

A **`filter string`** is also specified so that only messages with the message property **`office='New York'`** are diverted. All other messages continue to be routed to their usual address. The filter string is optional; if it is not specified, all messages are diverted.

The transformer class is also optional. When specified, transformation is executed for each matching message. This allows you to change the message body or properties before the message is diverted. The example above uses a transformation to add a header recording the time the divert occurred.

As a whole, the above example diverts 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.

33.2. NON-EXCLUSIVE DIVERTS

Non-exclusive diverts forward a copy of a message to a new address, allowing the original message to continue to the previous address. They can be thought of as splitting the message flow.

Non-exclusive diverts are configured similarly to exclusive diverts, with an optional filter and transformer, like so:

```
<divert name="order-divert">
   <address>jms.queue.orders</address>
   <forwarding-address>jms.topic.spyTopic</forwarding-address>
   <exclusive>false</exclusive>
</divert>
```

The **`order-divert`** example copies every message sent to the **`jms.queue.orders`** address (a JMS Queue called **`orders`**) and forwards the copy to a local address **`jms.topic.SpyTopic`** (a JMS Topic called **`SpyTopic`**).

CHAPTER 34. 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 due to network failure, the bridge will retry connecting to the target until it comes back on line. When it comes back on line 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 35, *Duplicate Message Detection*).

NOTE

Although they have similar function, do not 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 is always preferable to use a core bridge where possible. Core bridges use duplicate detection to provide *once and only once* guarantees. To provide the same guarantee using a JMS bridge, an XA which has a higher overhead and is more complex to configure would need to be used.

34.1. CONFIGURING BRIDGES

Bridges are configured in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**. An example follows (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, many of the defaults may be used, therefore it will not be necessary to specify them all explicitly.

The parameters are described in the following list.

Core Bridge Parameters

`name`

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 is a mandatory parameter.

The queue must already exist by the time the bridge is instantiated at start-up.

NOTE

If using JMS then normally the JMS configuration (***`JBOSS_DIST`*** **`/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-jms.xml`**) is loaded after the core configuration file **`<`*****`JBOSS_HOME`*****`>/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-configuration.xml`** is loaded. If the bridge is consuming from a JMS queue then ensure that the JMS queue is also deployed as a core queue in the core configuration. Refer to the bridge example for an example of this.

`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, 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 12, *Filter Expressions*.

`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 allows the opportunity to transform the message header or body before forwarding

`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 a multiplier to apply to the time since the last retry in order to compute the time to the next retry.

This allows an *exponential backoff* between retry attempts to be implemented.

For example:

If **`retry-interval`** is set to **`1000`** ms and **`retry-interval-multiplier`** is set to **`2.0`**, then, if the first reconnect attempt fails, there will be a wait of **`1000`** ms, then **`2000`** ms and 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 fail-over 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 fail-over onto its backup. If the bridge connector has no backup server configured, this parameter has no effect.

This parameter is useful when occasionally a bridge configured with a live and a backup target server is required, but fail-over to the backup is not required if the live server is taken down temporarily for maintenance.

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 the server crashes, when the bridge resumes it will resend unacknowledged messages. This might result in duplicate messages being sent to the target server. 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 (Refer to Chapter 35, *Duplicate Message Detection* 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 32, *Client Reconnection and Session Reattachment*.

WARNING

When using the bridge to forward messages from a queue which has a max-size-bytes set it is 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 on connectors and their configuration, refer to Chapter 14, *Configuring the Transport*.

The **`connector-ref`** element can be configured with two attributes:

- **`connector-name`**. This references the name of a connector defined in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**. It represents the connector that the bridge will fail-over to if it detects that the live server connection has failed. If this is specified and **`failover-on-server-shutdown`** is set to **`true`** then it will also attempt fail-over 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** is 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** is used.

CHAPTER 35. DUPLICATE MESSAGE DETECTION

HornetQ includes powerful automatic duplicate message detection, filtering out duplicate messages without having to code 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 is not possible to distinguish between 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 satisfactorily. If it decides to resend the last message, it may 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 doubly booked, which is an undesirable situation.

Sending the message(s) in a transaction is also not the correct solution. If the server or connection fails while the transaction commit is being processed, it is indeterminate whether the transaction was successfully committed or not.

To solve these issues HornetQ provides automatic duplicate messages detection for messages sent to addresses.

35.1. USING DUPLICATE DETECTION FOR MESSAGE SENDING

Enable duplicate message detection for sent messages: set a special property on the message to a uniquely created value. When the target server receives the message it will check whether that property is set. If it is, the target server will check its memory cache whether a message with the value of the header has already been received. If it has previously received a message with the same value it will ignore the message.

NOTE

Using duplicate detection to move messages between nodes can give the same *once and only once* delivery guarantees as when an XA transaction is used to consume messages from source and send them to the target, but with less overhead and much easier configuration than using XA.

If sending messages in a transaction it is not necessary to set the property for *every* message sent in that transaction, set it once in the transaction. If the server detects a duplicate message for any message in the transaction, it will ignore the entire transaction.

The name of the property set is given by the value of **`org.hornetq.api.core.HDR_DUPLICATE_DETECTION_ID`**, which is **`_HQ_DUPL_ID`**.

The value of the property can be of type **`byte[]`** or **`SimpleString`** if using the core API. If 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.

An example of setting the property using the core API follows:

```
...

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 is 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);

...
```

35.2. 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**. If this is set to **`true`** then each id will be persisted to permanent storage as it is received. The default value for this parameter is **`true`**.

NOTE

When choosing a size for the duplicate id cache, be sure to set it to a size large enough to prevent the previously sent messages from being overwritten.

35.3. DUPLICATE DETECTION AND BRIDGES

Core bridges can be configured to automatically add a unique duplicate id value (if there is not already one in the message) before forwarding the message to its 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, set the ***`use-duplicate-detection`*** to **`true`** when configuring a bridge in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**.

The default value for this parameter is **`true`**.

For more information on core bridges and how to configure them, refer to Chapter 34, *Core Bridges*.

35.4. DUPLICATE DETECTION AND CLUSTER CONNECTIONS

Cluster connections internally use core bridges to move messages reliable between nodes of the cluster. Consequently cluster connections can also be configured to insert the duplicate id header for each message moved using internal bridges.

To configure a cluster connection to add the duplicate id header, set the ***`use-duplicate-detection`*** to **`true`** when configuring a cluster connection in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/hornetq-configuration.xml`**.

The default value for this parameter is **`true`**.

For more information on cluster connections and how to configure them, refer to Chapter 36, *Clusters*.

35.5. DUPLICATE DETECTION AND PAGING

HornetQ uses duplicate detection when paging messages to storage. If a message is depaged from storage once upon server failure, the scenario where multiple messages are depaged resulting in duplicate delivery is prevented.

Paging, and how to configure it, is discussed in detail in Chapter 22, *Paging*.

CHAPTER 36. CLUSTERS

36.1. 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** to **`true`** (**`false`** by default).

IMPORTANT

You must enable clustering on those profiles that already contain a **`/hornetq`** directory, which is all profiles excluding **`Minimal`** and **`Web`**. Those profiles not containing a **`/hornetq`** directory do not natively contain the correct components to support a cluster.

The cluster is formed by each node declaring *cluster connections* to other nodes in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**. When a node forms a cluster connection to another node, internally it creates a *core bridge* (as described in Chapter 34, *Core Bridges*) connection between it and the other node, this is done transparently behind the scenes - you do not need 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. Common topologies are discussed later in this chapter.

Client side load balancing is discussed, where client connections can be balanced across the nodes of the cluster. Message redistribution where HornetQ will redistribute messages between nodes to avoid starvation is also covered.

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.

WARNING

If you start a JBoss Enterprise Application Platform instance with a HornetQ server bound to 'localhost', the HornetQ instance could form a cluster with another HornetQ instance on the same network. Binding to localhost does not provide cluster isolation for HornetQ servers.

To correctly isolate clusters, refer to Section 36.2.1, “Broadcast Groups” to correctly configure the broadcast and discovery addresses of each server.

36.2. SERVER DISCOVERY

Server discovery is a mechanism by which servers can propagate their connection details to:

- 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.
- 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 User Datagram Protocol (UDP) multicast to broadcast server connection settings. If UDP is disabled on your network you will not be able to use this, and will have to specify servers explicitly when setting up a cluster or using a messaging client.

36.2.1. 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, refer to Chapter 14, *Configuring the Transport*.

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 **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**. There can be many broadcast groups per HornetQ server. All broadcast groups must be defined in a **broadcast-groups** element.

Let us take a look at an example broadcast group from **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**:

```
<broadcast-groups>
   <broadcast-group name="my-broadcast-group">
      <local-bind-address>172.16.9.3</local-bind-address>
      <local-bind-port>5432</local-bind-port>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <broadcast-period>2000</broadcast-period>
      <connector-ref>netty</connector-ref>
   </broadcast-group>
</broadcast-groups>
```

Some of the broadcast group parameters are optional and the defaults will normally be used, but all are specified in the above example for clarity. Each are covered below.

Broadcast Group Parameters

name

Each broadcast group in the server must have a unique name in the server.

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. This parameter is always specified in conjunction with **`local-bind-address`**.

If you are behind a firewall you can utilize ***`local-bind-address`*** and ***`local-bind-port`*** to specify a static host and port. However, it is highly unlikely that a cluster would be configured with server instances outside a firewall which would need to communicate with the server which are behind the firewall.

`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 **`2000`** milliseconds.

`connector-ref`

This specifies the connector and optional backup connector that will be broadcast (see Chapter 14, *Configuring the Transport* for more information on connectors).

36.2.2. Discovery Groups

While the broadcast group defines how connector information is broadcast 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 discover what servers in the cluster they can connect to.

Although a discovery group will always accept broadcasts, its current list of available live and backup servers is only ever used when an initial connection is made, from then server discovery is done over the normal HornetQ connections.

36.2.3. Defining Discovery Groups on the Server

For cluster connections, discovery groups are defined in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/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 us look at an example:

```
<discovery-groups>
   <discovery-group name="my-discovery-group">
      <local-bind-address>172.16.9.7</local-bind-address>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

Each parameter of the discovery group is considered as follows:

Discovery Group Parameters

`name`

Each discovery group must have a unique name per server.

`local-bind-address`

If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group only listens on a specific interface. To do this you can specify the interface address with this parameter. This parameter is optional.

`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 server's 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).

36.2.4. Discovery Groups on the Client Side

Let us 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 are using JMS or the core API.

36.2.4.1. Configuring client discovery using JMS

If you are using JMS and you are 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 ***`JBOSS_DIST`*****`/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-jms.xml`**. Let us 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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 are using JMS, but you are not using JNDI to lookup a connection factory - you are instantiating the JMS connection factory directly then you can specify the discovery group parameters directly when creating the JMS connection factory. Here is 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 **`setDiscoveryInitialWaitTimeout()`**. If the connection factory is used immediately after creation then it may not have had enough time to receive 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 **`10000`** milliseconds.

36.2.4.2. Configuring client discovery using Core

If you are using the core API to directly instantiate **`ClientSessionFactory`** instances, then you can specify the discovery group parameters directly when creating the session factory. Here is an example:

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;
SessionFactory factory = HornetQClient.createClientSessionFactory
  (groupAddress, groupPort);
ClientSession session1 = factory.createClientSession(...);
ClientSession session2 = factory.createClientSession(...);
```

The **`refresh-timeout`** can be set directly on the session factory by using the setter method **`setDiscoveryRefreshTimeout()`** if you want to change the default value.

There is also a further parameter settable on the session factory using the setter method **`setDiscoveryInitialWaitTimeout()`**. If the session factory is used immediately after creation then it may not have had enough time to receive 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 **`10000`** milliseconds.

36.3. 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 us take a simple example of a cluster of four nodes A, B, C, and D arranged in a *symmetric cluster* (described in Section 36.7.1, "Symmetric cluster"). A queue called **`OrderQueue`** is deployed on each node of the cluster.

A client Ca is connected to node A, sending orders to the server. Also, order processor clients Pa, Pb, Pc, and Pd are connected to each of the nodes A, B, C, D. If no cluster connection was defined on node A, as order messages arrive on node A they will all end up in the **`OrderQueue`** on node A, so they will get consumed by the order processor client attached to node A, Pa.

If a cluster connection on node A is defined, as ordered messages arrive on node A, they are distributed in a round-robin fashion between all the nodes of the cluster, instead of all of them going into the local **`OrderQueue`** instance. 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.

36.3.1. Configuring Cluster Connections

Cluster connections group servers into clusters so that messages can be load balanced between the nodes of the cluster. Typical cluster connections are defined in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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, applies 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.

Be careful not to have multiple cluster connections with overlapping values of **`address`**, (for example, "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.

- **`discovery-group-ref`**. This parameter determines which discovery group is used to obtain the list of other servers in the cluster to which this cluster connection will make connections.

- **`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 processed in a round robin style 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 and 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. This is covered 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.

- **`min-large-message-size`**. This parameter determines the size threshold above which a message will be split into multiple packages when sent over the cluster. This parameter is optional and the default is **`100 kB`**.

- **`reconnect-attempts`**. This parameter determines the number of times the system will try to connect a node on the cluster. If the max-retry is achieved this node will be considered permanently down and the system will stop routing messages to it. This parameter is optional and the default is **`-1`** (infinite retries).

- **`retry-interval`**. 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 34, *Core Bridges*).

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, refer to Chapter 35, *Duplicate Message Detection*.

  This parameter is optional and has a default value of **`true`**.

36.3.2. 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 **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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.

36.4. 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.

It is possible to 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 do not specify a policy then the default will be used which is **`org.hornetq.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy`**.

If you are using JMS, and you are using JNDI on the server to put your JMS connection factories into JNDI, then you can specify the load balancing policy directly in the **`JBOSS_DIST/jboss-as/server/<PROFILE>/deploy/hornetq/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>
   <ha>true</ha>
   <connection-load-balancing-policy-class-name>

org.hornetq.api.core.client.loadbalance.RandomConnectionLoadBalancingPolic
y
   </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 are using JMS but you are 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.MyLoadBalan
cingPolicy");
```

If you are 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.

36.5. SPECIFYING MEMBERS OF A CLUSTER EXPLICITLY

Sometimes UDP is not enabled on a network so it is 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. This is done as follows:

36.5.1. Specify List of Servers on the Client Side

This differs depending on whether you are using JMS or the Core API.

36.5.1.1. Specifying List of Servers using JMS

If using JMS and 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 ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml**. Let us 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 **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/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 refer to Chapter 14, *Configuring the Transport*.

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 are using JMS but you are not using JNDI then you can also specify the list of [connector, backup connector] pairs directly when instantiating the **HornetQConnectionFactory**. Here is 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();
```

The above snippet creates a list of pairs of **`TransportConfiguration`** objects. Each **`TransportConfiguration`** object contains knowledge of how to make a connection to a specific server.

Create a **`HornetQConnectionFactory`** instance, 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.

36.5.1.2. Specifying List of Servers using the Core API

Specify the list of servers directly when creating the **`ClientSessionFactory`** instance as in the following 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 session1 = factory.createClientSession(...);

ClientSession session2 = factory.createClientSession(...);
```

The above snippet creates 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, refer to Chapter 14, *Configuring the Transport*.

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.

36.5.2. Specifying a Static Cluster Server List

It is possible to define a symmetric cluster and not use static server discovery so each node can in turn discover available nodes. Configuring each cluster connection to have explicit knowledge of all the other nodes in the cluster is required.

IMPORTANT

Fail-over is not supported for clusters defined using a static cluster server list. To support fail-over between cluster nodes, the nodes must be configured to use a discovery group.

Task: Specify Cluster Server List without Auto Discovery

Complete this task to specify a static cluster server list instead of using server auto discovery.

Prerequisites

- The **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** file open, ready to add directives.
- Understand the **hornetq-configuration.xml** configuration directives, as detailed in Section A.1.1, “hornetq-configuration.xml”.

1. **Define Connectors**
   In the **hornetq-configuraton.xml** file, insert a <connectors> directive block defining the remoting connector factory, the names of connectors, and the ports each connector will use.

   Each connector must use a unique port.

```
<connectors>
  <connector name="netty-connector">
    <factory-class>
     org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
    </factory-class>
  <param key="port" value="5445"/>
</connector>
<!-- connector to the server1 -->
  <connector name="server1-connector">
    <factory-class>
     org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
    </factory-class>
    <param key="port" value="5446"/>
  </connector>
<!-- connector to the server2 -->
  <connector name="server2-connector">
    <factory-class>
     org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
    </factory-class>
    <param key="port" value="5447"/>
  </connector>
</connectors>
```

2. **Define Cluster Connection**
   Insert a <cluster-connection> directive block. The block must contain mandatory clustering directives, and the <connector-ref> directives set in the previous step. The <connector-ref> directives use the name attribute set in the <connector> directives.

```
<cluster-connections>
  <cluster-connection name="my-cluster">
    <address>jms</address>
    <connector-ref>netty-connector</connector-ref>
    <retry-interval>500</retry-interval>
    <use-duplicate-detection>true</use-duplicate-detection>
    <forward-when-no-consumers>true</forward-when-no-consumers>
    <max-hops>1</max-hops>
    <static-connectors>
      <connector-ref>server1-connector</connector-ref>
      <connector-ref>server2-connector</connector-ref>
    </static-connectors>
  </cluster-connection>
</cluster-connections>
```

3. **Result**
   The cluster is now defined with the directives required for server discovery using explicit server names.

36.6. MESSAGE REDISTRIBUTION

Another important part of clustering is message redistribution. Earlier, it was demonstrated how server side message load balancing round robins or directs messages to all nodes across the cluster. If **`forward-when-no-consumers`** is false, messages will not be forwarded to nodes which do not have matching consumers. This ensures that messages do not arrive on a queue which has no consumers to consume them, however there is a situation it does not 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 will not get consumed and a *starvation* situation will be created.

Message redistribution addresses this and HornetQ can be configured to automatically *redistribute* messages from queues which have no consumers to other nodes in the cluster which *do* have matching consumers.

Message redistribution can be configured to act immediately after the last consumer on a queue is closed, or to wait for a configurable delay after the last consumer on a queue is closed before redistributing. By default, message redistribution is enabled with a delay of 60000 milliseconds (1 minute).

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, refer to Chapter 23, *Queue Attributes*.

An address settings snippet from **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** showing how message redistribution is enabled for a set of queues follows:

```
<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 11, *Understanding the HornetQ Wildcard Syntax*).

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.

It often makes sense to introduce a delay before redistributing as it is a common case that a consumer closes but another one quickly is created on the same queue, in such a case you probably do not want to redistribute immediately since the new consumer will arrive shortly.

36.7. CLUSTER TOPOLOGIES

HornetQ clusters can be connected together in many different topologies, let us consider the two most common ones here:

36.7.1. Symmetric cluster

A symmetric cluster is probably the most common cluster topology, and you will be familiar with if you have 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: 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.

36.7.2. 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, cluster connections are only desired in one direction since messages are only moving from node A->B->C and never from C->B->A.

For this topology, 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.

CHAPTER 37. HIGH AVAILABILITY AND FAIL-OVER

High availability is defined as the ability for the system to continue functioning after failure of one or more of the servers.

A part of high availability is *fail-over* which is defined as the ability for client connections to migrate from one server to another in the event of server failure so that client applications can continue to operate.

WARNING

HornetQ requires a stable, reliable connection to the file system where its journal is located. If connectivity between HornetQ and the journal is lost and later re-established, an I/O error for messaging will occur. This error is considered a "major event" and requires manual intervention with the messaging system in order to recover (i.e. the messaging system will need to be restarted). If this occurs on a cluster node, other nodes will take on the load of the failed node, providing they have been configured to do so.

37.1. 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 fail-over occurs.

Before fail-over, only the live server is serving the HornetQ clients while the backup servers remain passive or awaiting to become a backup server. When a live server crashes or is brought down in the correct mode, the backup server currently in passive mode will become live and another backup server will become passive. If a live server restarts after a fail-over then it will have priority and be the next server to become live when the current live server goes down, if the current live server is configured to allow automatic fail back then it will detect the live server coming back up and automatically stop.

37.1.1. HA modes

HornetQ provides only *shared store* in this release.

NOTE

Only persistent message data will survive fail-over. Non-persistent message data is lost after fail-over occurs.

37.1.2. Shared Store

When using a shared store, both live and backup servers share the *same* entire data directory using a shared file system. This means the paging directory, journal directory, large messages and binding journal.

When fail-over occurs and the backup server takes over, it will load the persistent storage from the shared file system and clients can connect to it.

IMPORTANT

HornetQ HA supports shared store on GFS2 on SAN.

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). Do not use NFS mounts to store any shared journal when using NIO (non-blocking I/O). Also consider that NFS is not ideal due to the data transfer rate of this standard.

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 the highest performance during normal operation is required and there is access to a fast SAN, and a slightly slower fail-over is acceptable (depending on amount of data), shared store high availability is recommended.

37.1.2.1. Configuration

To configure the live and backup server to share their store, configure both **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** files on each node:

```
<shared-store>true</shared-store>
```

Additionally, the backup server must be flagged explicitly as a backup:

```
<backup>true</backup>
```

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 13.3, “Configuring the message journal”)

The Live and Backup pair must have a cluster connection defined, even if the pair is not part of a cluster. The Cluster Connection info defines how backup servers announce their presence to a live server or any other nodes in the cluster. Refer to Chapter 36, *Clusters* for details on how to configure this.

37.1.2.2. Failing Back to Live Server

After a live server has failed and a backup has taken over its duties, you may want to restart the live server and have clients fail back. To do this, restart the original live server and stop the new live server. You can do this by terminating the process itself or waiting for the server to shut down.

It is also possible to cause fail-over to occur on normal server shutdown, to enable this set the following property to true in **<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml**:

```
<failover-on-shutdown>true</failover-on-shutdown>
```

You can force the new live server to shutdown when the old live server comes back up allowing the original live server to take over automatically by setting the following property in

***<JBOSS_HOME>*/jboss-as/server/*<PROFILE>*/deploy/hornetq/hornetq-configuration.xml** as follows:

```
<allow-failback>true</allow-failback>
```

37.2. FAIL-OVER MODES

HornetQ defines two types of client fail-over:

- Automatic client fail-over
- Application-level client fail-over

HornetQ provides transparent automatic reattachment of connections to the same server (for example, in case of transient network problems). This is similar to fail-over, except the connection is reconnecting to the same server. More information on this topic is discussed in Chapter 32, *Client Reconnection and Session Reattachment*.

During fail-over, if the client has consumers on any non persistent or temporary queues, those queues will be automatically recreated during fail-over on the backup node, since the backup node will not have any knowledge of non persistent queues.

37.2.1. Automatic Client fail-over

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 fail-over, 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 15, *Detecting Dead Connections*. If the client does not receive data in good time, it will assume the connection has failed and attempt fail-over.

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, refer to Section 36.2, “Server discovery”. Alternatively, the clients can explicitly specify pairs of live-backup server as explained in Section 36.5.2, “Specifying a Static Cluster Server List”.

To enable automatic client fail-over, the client must be configured to allow non-zero reconnection attempts (as explained in Chapter 32, *Client Reconnection and Session Reattachment*).

Sometimes it is desirable for a client to fail-over onto a backup server even if the live server is just cleanly shutdown rather than having crashed or the connection failed. To configure this set the property **FailoverOnServerShutdown** to true either on the **HornetQConnectionFactory** if using JMS or in the ***JBOSS_DIST*/jboss-as/server/*<PROFILE>*/deploy/hornetq/hornetq-jms.xml** file when defining 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* fail-over to a backup server if the live server is shutdown cleanly.

NOTE

Cleanly shutting down the server will not trigger fail-over on the client by default. For the client to fail-over when its server is cleanly shutdown, set the property ***`FailoverOnServerShutdown`*** to true.

Using **`Ctrl`**+**`C`** (in a Linux terminal) causes the server to cleanly shut down, so client fail-over is not triggered unless this property is correctly configured.

By default fail-over will only occur after at least one connection has been made to the live server. Applying this logic practically means that fail-over will not occur if the client fails to make an initial connection to the live server. The client will retry connecting to the live server according to the reconnect-attempts property and fail after this number of attempts.

In some cases, you may want the client to automatically try the backup server it fails to make an initial connection to the live server. In this case, set the property **`FailoverOnInitialConnection`**, or **`failover-on-initial-connection`** in XML, on the **`ClientSessionFactoryImpl`** or **`HornetQConnectionFactory`**. The default value for this parameter is **`false`**.

NOTE

HornetQ does not replicate full server state between live and backup servers. When the new session is automatically recreated on the backup it will not have any knowledge of messages already sent or acknowledged in that session. Any in-flight sends or acknowledgments at the time of fail-over might also be lost.

37.2.1.1. Handling Blocking Calls During fail-over

If the client code is in a blocking call to the server, waiting for a response to continue its execution, when fail-over 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 fail-over 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.

37.2.1.2. Handling fail-over 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 acknowledgments have not been lost during the fail-over.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit 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. There is no need to manually rollback the session - it is already rolled back. The user can then just retry the transactional operations again on the same session.

If fail-over 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 enable duplicate detection (Chapter 35, *Duplicate Message Detection*) 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 fail-over, 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.

37.2.1.3. Handling fail-over With Non Transactional Sessions

If the session is non transactional, messages or acknowledgments can be lost in the event of fail-over.

To provide *once and only once* delivery guarantees for non transacted sessions too, enabled duplicate detection, and catch unblock exceptions as described in Section 37.2.1.1, “Handling Blocking Calls During fail-over”

37.2.2. Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: **`java.jms.ExceptionListener`**. For more information about ExceptionListener, refer to the Oracle javax.jms Javadoc.

The HornetQ core API also provides a similar feature in the form of the class **`org.hornet.core.client.SessionFailureListener`**

Any JMS **`ExceptionListener`** or Core **`SessionFailureListener`** instance will always be called by HornetQ in the event of connection failure, irrespective of whether the connection was successfully failed over, reconnected or reattached.

37.2.3. Application-Level fail-over

In some cases automatic client fail-over may not be desirable, and you may prefer to handle any connection failure yourself, and code your own manual reconnection logic in your own failure handler. This defined as *application-level* fail-over, since the fail-over is handled at the user application level.

To implement application-level fail-over, if using JMS, set an **`ExceptionListener`** class on the JMS connection. The **`ExceptionListener`** will be called by HornetQ in the event that connection failure is detected. In **`ExceptionListener`**, close the 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 to ensure that the new connection factory is looked up from a different server.

If using the core API, the procedure is very similar: set a **`SessionFailureListener`** on the core **`ClientSession`** instances.

37.3. FENCING

Fencing nodes in a cluster is implemented to isolate a malfunctioning node from the rest of a cluster. This is important to prevent the scenario where a malfunctioning node assumes the rest of a the cluster is in error, and tries to fail over the cluster to the malfunctioning node. This scenario could create a race condition and cause extensive data corruption.

While HornetQ can operate in a fenced environment, configuration varies depending on what fencing agent you choose, and how the fencing agent is configured.

If your server cluster is running on Red Hat Enterprise Linux, Red Hat provides fencing support through the High Availability Add-On. You must have an entitlement to use this product in the Customer Support Portal .

For detailed information about configuring fencing using the High Availability Add-On, refer to the Red Hat Enterprise Linux *Cluster Administration Guide* for the version of Red Hat Enterprise Linux installed on your cluster infrastructure. The sections "Configuring Fence Devices", and "Configuring Fencing for Cluster Members" will help you configure fencing correctly.

CHAPTER 38. COLOCATED AND DEDICATED SYMMETRICAL CLUSTER CONFIGURATION

Read this chapter to configure HornetQ *live backup groups* within JBoss Enterprise Application Platform. HornetQ only supports a shared store for live backup nodes, therefore the chapter covers configuring the live backup groups in this way.

Live Backup Group

An instance of HornetQ running on JBoss Enterprise Application Platform that is configured to fail over to a specified group of HornetQ instances.

HornetQ live backup groups are configured using one of the following topologies:

Colocated

Topology containing one live, and at least one back-up server running concurrently. Each backup node belongs to a live node on another JBoss Enterprise Application Platform instance.

Dedicated

Topology containing one live and at least one backup server. Only one server can run at any given time.

38.1. COLOCATED SYMMETRICAL LIVE AND BACKUP CLUSTER

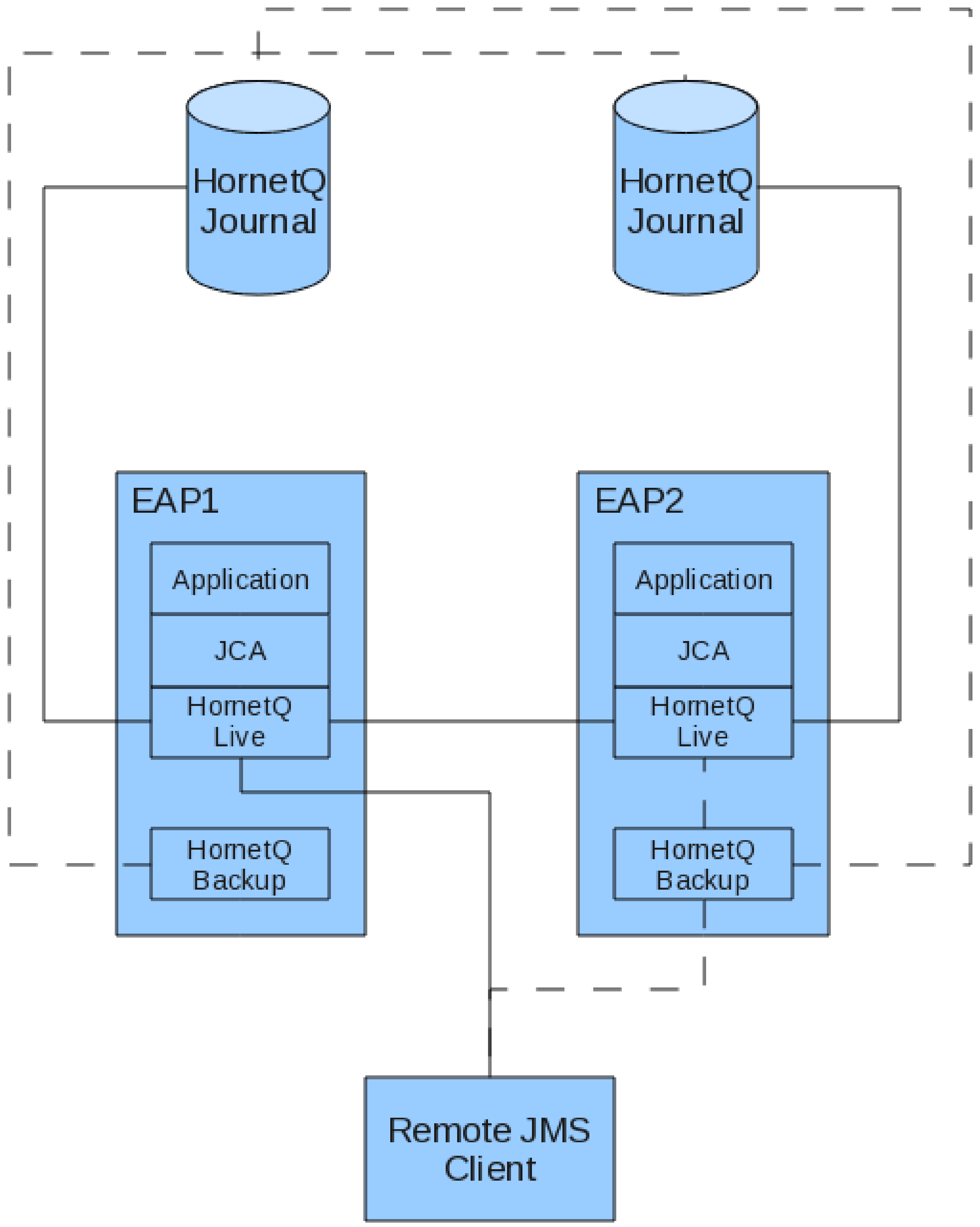
NOTE

JBoss Enterprise Application Platform ships with an example configuration for this topology, located in **`$JBOSS_HOME/extras/hornetq/resources/examples/symmetric-cluster-with-backups-colocated`**. The **`readme`** in this directory provides basic configuration required to run the example.

The colocated symmetrical topology contains an operational live node, and one or more backup nodes. Each backup node belongs to a live node on another JBoss Enterprise Application Platform instance.

In a simple cluster of two JBoss Enterprise Application Platform instances, each JBoss Enterprise Application Platform instance would have a live server and one backup server. as described in Example 38.1, “Two Instance Configuration”.

Example 38.1. Two Instance Configuration



The continuous lines in Example 38.1, “Two Instance Configuration”show the state of the cluster before fail over occurs. The dotted lines show the state of the cluster after fail over has occurred.

Before fail over occurs, the two live servers are connected forming a cluster. Each live server is connected to its local applications through J2EE Connector Architecture (JCA). The remote clients residing are connected to their respective live servers.

When fail over occurs, the HornetQ Backup connects to the still available live server (which happens to be in the same virtual machine) and takes over as the live server in the cluster. Any remote clients also fail over.

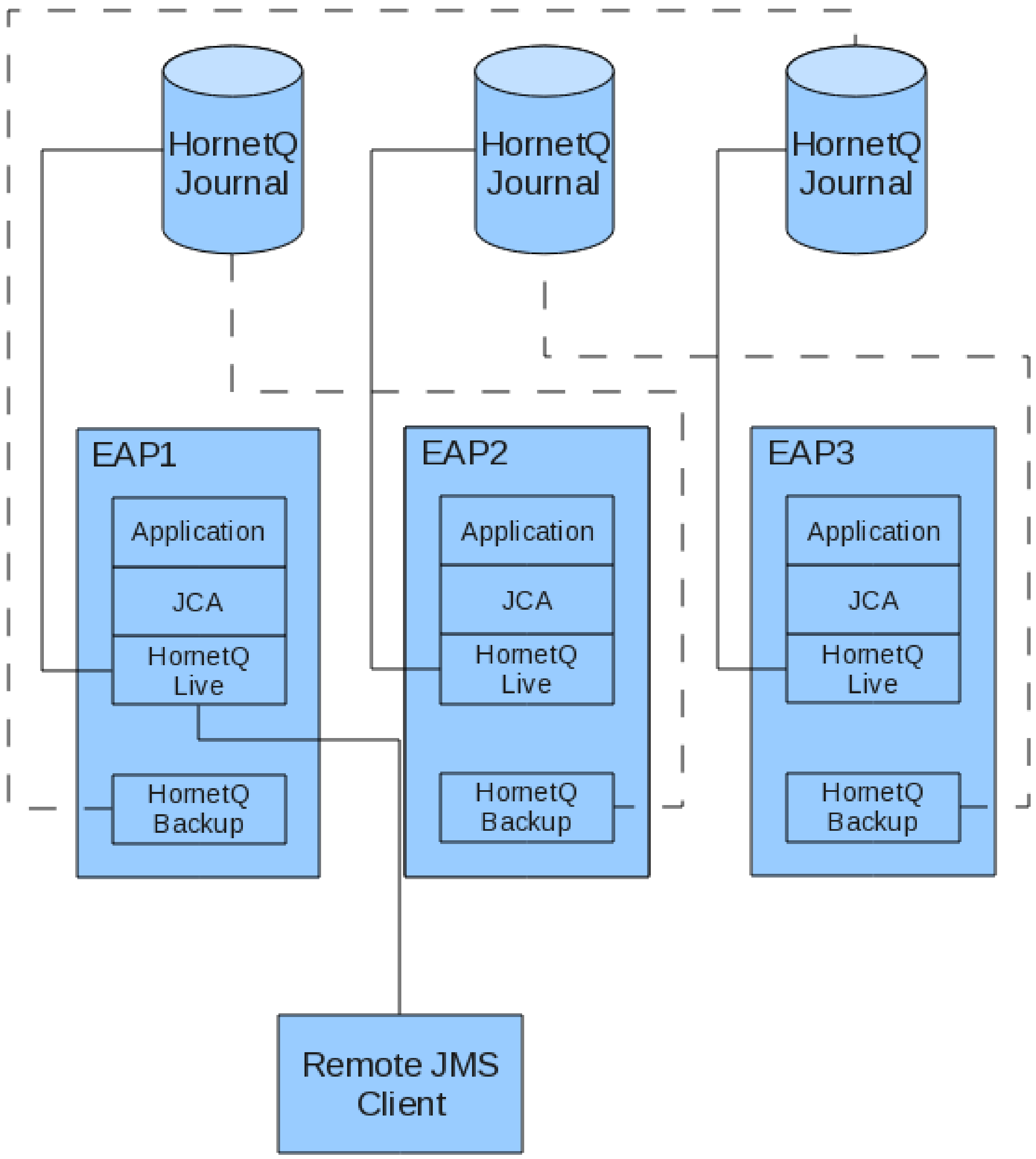
Depending on what consumers and producers, and Message Driven Beans (MDBs) are available on each node, messages are distributed between the nodes to satisfy Java Message Service (JMS) requirements. For example, if a producer is sending messages to a queue on a backup server with no consumers, the messages will be distributed to a live node advertising the required consumers.

Example 38.2, “Three Instance Configuration” is slightly more complex. It extends the same configuration in Example 38.1, “Two Instance Configuration” and adds a third live and backup HornetQ instance.

NOTE

The live cluster connections between each server have been removed to make the diagram clearer. All live servers form a cluster in the example.

Example 38.2. Three Instance Configuration



In this example, the instance contains three separate JBoss Enterprise Application Platform servers, with a live and backup HornetQ instance in each server.

Three node topology enables you to configure two backup instances for each server, which are shared across any of the three servers in the live backup group in case of a fail over event.

While it is possible to configure multiple live backup groups for a server, one backup per live instance is considered sufficient for most deployments.

38.1.1. Colocated Live Server

IMPORTANT

If nodes can not discover each other, verify that you have configured the firewall and UDP ports correctly. The network configuration should allow nodes on a cluster to communicate with each other.

Follow the procedures to configure a Colocated Live Server instance.

Procedure 38.1. Create Live Server Profile

You must create a copy of the **`production`** profile to customize the live server configuration.

IMPORTANT

Always copy an included profile rather than editing it directly for your custom profile. If you make a critical mistake during configuration, you can fall back to the base configuration at any time.

1. Navigate to **`$JBOSS_HOME/server/`**
2. Copy the **`production`** profile, and rename it to **`HornetQ_Colocated`**

Procedure 38.2. Configure Shared Store and Journaling

Follow this procedure to specify HornetQ must use a shared store for fail over, and define the location of the journal files each HornetQ instance in the live backup group uses.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/`**
2. Open
3. Add the <shared-store> element as a child of the <configuration> element.

   ```
   <shared-store>true</shared-store>
   ```

4. Ensure the bindings, journal, and large messages path locations are set to a location the live backup group can access.

   You can set absolute paths as the example describes, or use the JBoss parameters that exist in the configuration file.

   If you choose the parameter option, and you do not use the default paths that these parameters resolve to, you must specify the path your bindings, journal, and large messages reside in each time you start the server.

   ```
   <large-messages-directory>/media/shared/data/serverA/large-
   messages</large-messages-directory>

   <bindings-directory>/media/shared/data/serverA/bindings</bindings-
   directory>

   <journal-directory>/media/shared/data/serverA/journal</journal-
   directory>
   ```

```
<paging-directory>/media/shared/data/ServerA/paging</paging-
directory>
```

NOTE

Ensure you specify paths that are accessible to the live backup groups on your network.

NOTE

Change *ServerA* to the name suitable to your server instance.

By default, JMS clients will not fail over if a live server is shut down gracefully. Depending on the connection factory settings, a client will either fail, or try to reconnect to the live server.

If you need clients to fail over on a normal server shutdown, you must alter the file according to Procedure 38.3, “Configure JMS Client Graceful Shutdown”.

Procedure 38.3. Configure JMS Client Graceful Shutdown

Follow this procedure to configure how JMS clients re-establish a connection if a server is shut down gracefully.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Specify the <fail-over-on-shutdown> element in the area near the journal directory configuration in Procedure 38.2, “Configure Shared Store and Journaling”.

   ```
   <failover-on-shutdown>true</failover-on-shutdown>
   ```

   NOTE

   You are not constrained where you put the element in the **`hornetq-configuration.xml`** file, however it is easier to find the less detailed settings if they are all located at the top of the file.

4. Save and close the file.

NOTE

If you have set <fail-over-on-shutdown> to false (the default setting) but still want fail over to occur, terminate the server process directly, or call **`forceFailover`** through the JMX Console or the Admin Console on the core server object.

The connection factories used by the client must be configured to be Highly Available. This is done by configuring connection factory attributes in the ***`JBOSS_DIST`*****`/jboss-as/server/<`*****`PROFILE`*****`>/deploy/hornetq/hornetq-jms.xml`**.

Procedure 38.4. Configure HA Connection Factories

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq/`**
2. Open **`hornetq-jms.xml`**.
3. Add the following attributes and values as specified below.

   <ha>true</ha>

   Specifies the client must support high availability, and must always be true for fail over to occur.

   <retry-interval>1000</retry-interval>

   Specifies how long the client must wait (in milliseconds) before it can reconnect to the server.

   <retry-interval-multiplier>1.0</retry-interval-multiplier>

   Specifies the multiplier <retry-interval> used for each subsequent reconnection pauses. By setting the value to **`1.0`**, the retry interval is the same for each client reconnection request.

   <reconnect-attempts>-1</reconnect-attempts>

   Specifies how many reconnect attempts a client should make before failing. Setting **`-1`** means unlimited reconnection attempts.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration xmlns="urn:hornetq"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <connection-factory name="NettyConnectionFactory">
      <xa>true</xa>
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
         <entry name="/XAConnectionFactory"/>
      </entries>
      <ha>true</ha>

      <!-- Pause 1 second between connect attempts -->

      <retry-interval>1000</retry-interval>

      <!-- Multiply subsequent reconnect pauses by this multiplier.
This can be used
       to implement an exponential back-off. For our purposes we
just set to 1.0 so
       each reconnect pause is the same length -->

      <retry-interval-multiplier>1.0</retry-interval-multiplier>

      <!-- Try reconnecting an unlimited number of times (-1 means
unlimited) -->

      <reconnect-attempts>-1</reconnect-attempts>
   </connection-factory>

</configuration>
```

4. Define new queues in both master and backup nodes by adding one of the following configuration blocks to the specified file.

   For **`production/deploy/hornetq/hornetq-jms.xml`**

```
<queue name="testQueue">
   <entry name="/queue/testQueue"/>
   <durable>true</durable>
</queue>
```

   For **`production/deploy/`*`customName`*`-hornetq-jms.xml`**

NOTE

Ensure the file is well-formed from an XML validation perspective by ensuring the XML Namespace is present and correct in the file as specified.

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

   <queue name="testQueue">
      <entry name="/queue/testQueue"/>
      <durable>true</durable>
   </queue>

</configuration>
```

38.1.2. Colocated Backup Server

Read this section and the contained procedures to configure a colocated backup HornetQ server.

The backup server runs on the same JBoss Enterprise Application Platform instance as the live server configured in Section 38.1.1, “Colocated Live Server”. The important thing to understand is the backup server is the fail over point for a live server running on a different JBoss Enterprise Application Platform instance.

Likewise, the backup server instance does not service any Java EE components on the JBoss Enterprise Application Platform instance it is colocated on. When the live server fails over, any existing messages are redistributed within the live backup group, to service any remote clients connected to the live server at the time it became unavailable.

A backup HornetQ instance only needs a **`hornetq-jboss-beans.xml`** and a **`hornetq-configuration.xml`** configuration file. Any JMS components are created from the shared journal when the backup server becomes live (configured in Procedure 38.2, “Configure Shared Store and Journaling”).

Procedure 38.5. Create Backup Server

IMPORTANT

If nodes can not discover each other, verify that you have configured the firewall and UDP ports correctly. The network configuration should allow nodes on a cluster to communicate with each other.

You must set up the live server first as specified in Section 38.1.1, “Colocated Live Server”. After you have configured the live server, continue with this procedure.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Colocated/deploy/`**

2. Create a new directory called **`hornetq-backup1`**. Move into that directory.

3. Open a text editor and create a new file called **`hornetq-jboss-beans.xml`** in the **`hornetq-backup1`** directory.

4. Copy the following configuration into **`hornetq-jboss-beans.xml`**.

```
<?xml version="1.0" encoding="UTF-8"?>
   <deployment xmlns="urn:jboss:bean-deployer:2.0">
      <!-- The core configuration -->
      <bean name="BackupConfiguration"
class="org.hornetq.core.config.impl.FileConfiguration">
         <property
name="configurationUrl">${jboss.server.home.url}/deploy/hornetq-
backup1/hornetq-configuration.xml</property>
      </bean>
      <!-- The core server -->
      <bean name="BackupHornetQServer"
class="org.hornetq.core.server.impl.HornetQServerImpl">
         <constructor>
            <parameter>
               <inject bean="BackupConfiguration"/>
            </parameter>
            <parameter>
               <inject bean="MBeanServer"/>
            </parameter>
            <parameter>
               <inject bean="HornetQSecurityManager"/>
            </parameter>
         </constructor>
         <start ignored="true"/>
         <stop ignored="true"/>
      </bean>

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

```
        </bean>

</deployment>
```

5. Save and close the file.

The **hornetq-jboss-beans.xml** file in Procedure 38.5, “Create Backup Server” contains configuration worth exploring in more detail. The BackupConfiguration bean is configured to pick up the configuration in **hornetq-configuration.xml**. This file is created in the next procedure: Procedure 38.6, “Create Backup Server Configuration File”.

The HornetQ Server and JMS server beans are added after the BackupConfiguration.

NOTE

The names of the backup instance beans have been changed from those in the live server configuration. This is to prevent problems with beans sharing the same name. If you add more backup servers, you must rename these instances uniquely as well (for example: backup1, backup2).

The next task involves creating the backup server configuration according to Procedure 38.6, “Create Backup Server Configuration File”.

Procedure 38.6. Create Backup Server Configuration File

1. Navigate to **$JBOSS_HOME/server/HornetQ_Colocated/deploy/hornetq-backup1**
2. Open a text editor and create a new file called **hornetq-configuration.xml** in the **hornetq-backup1** directory.
3. Copy the following configuration into **hornetq-configuration.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="urn:hornetq"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">

   <jmx-domain>org.hornetq.backup1</jmx-domain>

   <clustered>true</clustered>

   <backup>true</backup>

   <shared-store>true</shared-store>

   <allow-failback>true</allow-failback>

   <file-deployment-enabled>true</file-deployment-enabled>

   <log-delegate-factory-class-name>
    org.hornetq.integration.logging.Log4jLogDelegateFactory
   </log-delegate-factory-class-name>

   <bindings-directory>
```

```
    /media/shared/data/hornetq-backup/bindings
   </bindings-directory>

   <journal-directory>/media/shared/data/hornetq-
backup/journal</journal-directory>

   <journal-min-files>10</journal-min-files>

   <large-messages-directory>
    /media/shared/data/hornetq-backup/largemessages
   </large-messages-directory>

   <paging-directory>/media/shared/data/hornetq-
backup/paging</paging-directory>

   <connectors>
      <connector name="netty-connector">
         <factory-class>
          org.hornetq.core.remoting.impl.netty.NettyConnectorFactory
         </factory-class>
         <param key="host"
value="${jboss.bind.address:localhost}"/>
         <param key="port"
value="${hornetq.remoting.netty.backup.port:5446}"/>
      </connector>

      <connector name="in-vm">
         <factory-class>
          org.hornetq.core.remoting.impl.invm.InVMConnectorFactory
         </factory-class>
         <param key="server-id" value="${hornetq.server-id:0}"/>
      </connector>
   </connectors>

   <acceptors>
      <acceptor name="netty">
         <factory-class>
          org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory
         </factory-class>
         <param key="host"
value="${jboss.bind.address:localhost}"/>
         <param key="port"
value="${hornetq.remoting.netty.backup.port:5446}"/>
      </acceptor>
   </acceptors>

   <broadcast-groups>
      <broadcast-group name="bg-group1">
         <group-address>231.7.7.7</group-address>
         <group-port>9876</group-port>
         <broadcast-period>1000</broadcast-period>
         <connector-ref>netty-connector</connector-ref>
      </broadcast-group>
   </broadcast-groups>

   <discovery-groups>
```

```
      <discovery-group name="dg-group1">
         <group-address>231.7.7.7</group-address>
         <group-port>9876</group-port>
         <refresh-timeout>60000</refresh-timeout>
      </discovery-group>
   </discovery-groups>

   <cluster-connections>
      <cluster-connection name="my-cluster">
         <address>jms</address>
         <connector-ref>netty-connector</connector-ref>
         <discovery-group-ref discovery-group-name="dg-group1"/>
      </cluster-connection>
   </cluster-connections>

   <security-settings>
      <security-setting match="#">
         <permission type="createNonDurableQueue" roles="guest"/>
         <permission type="deleteNonDurableQueue" roles="guest"/>
         <permission type="consume" roles="guest"/>
         <permission type="send" roles="guest"/>
      </security-setting>
   </security-settings>

   <address-settings>
   <!--default for catch all-->
      <address-setting match="#">
         <dead-letter-address>jms.queue.DLQ</dead-letter-address>
         <expiry-address>jms.queue.ExpiryQueue</expiry-address>
         <redelivery-delay>0</redelivery-delay>
         <max-size-bytes>10485760</max-size-bytes>
         <message-counter-history-day-limit>10</message-counter-
history-day-limit>
         <address-full-policy>BLOCK</address-full-policy>
      </address-setting>
   </address-settings>

</configuration>
```

4. Save and close the file.

The **`hornetq-configuration.xml`** file in Procedure 38.6, “Create Backup Server Configuration File” contains specific configuration which is discussed in hornetq-configuration.xml Configuration Points.

hornetq-configuration.xml Configuration Points

<jmx-domain>org.hornetq.backup1</jmx-domain>

Specifies the object name (in this case the backup server) in the Java Management Extensions (JMX) service. The default value is **`org.hornetq`**, however this name is already in use in other parts of HornetQ. You must change the name to a unique, system-wide name to avoid naming conflicts with the live server.

<clustered>true</clustered>

Specifies whether the server should join a cluster. This configuration is the same as the live server.

<backup>true</backup>

Specifies whether the server starts as a backup server, and not a live server. Specifying true sets the server to start as a backup server.

<shared-store>true</shared-store>

Specifies whether the server should reference a shared store for journaling. This configuration is the same as the live server.

<allow-failback>true</allow-failback>

Specifies whether the backup server automatically stops and returns to standby mode when the live server becomes available again. If set to `false`, the server must be stopped manually to trigger a return to standby mode.

<bindings-directory>, <journal-directory>, <large-messages-directory>, <paging-directory>

The paths in these elements must all resolve to the same paths the live server references. This ensures the backup server uses the same journaling files as the live server.

<connectors>

Two connectors are defined that allow clients to connect to the backup server once live: one connector for the netty connector factory (to allow client and server connections across different Virtual Machines); and one connector to allow the server to accept connections within the VM.

<acceptors>

The NettyAcceptorFactory is chosen here for VM compatibility.

<broadcast-groups>, <discovery-groups>, <cluster-connections>, <security-settings>, <address-settings>

The settings in these configuration blocks are standard settings.

Task: Create Configuration for Second Server Instance

Complete this task to configure the second server instance to cluster with the first server.

Prerequisites

- Procedure 38.5, “Create Backup Server”
- Procedure 38.6, “Create Backup Server Configuration File”

1. Navigate to **`<JBOSS_HOME> /server/`**.
2. Copy the **`HornetQ_Colocated`** directory, and rename it to **`HornetQ_Colocated_Second`**.
3. Rename **`<JBOSS_HOME>/server/HornetQ_Colocated_Second/hornetq-backup1/`** to **`<JBOSS_HOME>/server/HornetQ_Colocated_Second/hornetq-backup-serverA/`**
4. Open **`<JBOSS_HOME>/server/HornetQ_Colocated_Second/hornetq/hornetq-configuration.xml`**

5. For all parameters with data directories specified in **`hornetq-configuration.xml`**, change the data paths to **`/media/shared/data/hornetq-backup`**.

   For example change:

   <bindings-directory> /media/shared/data/serverA/bindings </bindings-directory>

   to <bindings-directory> /media/shared/data/hornetq-backup/bindings </bindings-directory>

6. Open **`<JBOSS_HOME>/server/HornetQ_Colocated_Second/hornetq-backup-serverA/hornetq-configuration.xml`**

7. For all parameters with data directories specified in **`hornetq-configuration.xml`**, change the data paths to **`/media/shared/data/serverA`**.

   For example change:

   <bindings-directory> /media/shared/data/hornetq-backup/bindings </bindings-directory>

   to <bindings-directory> /media/shared/data/serverA/bindings </bindings-directory>

38.2. DEDICATED SYMMETRICAL LIVE AND BACKUP CLUSTERS

NOTE

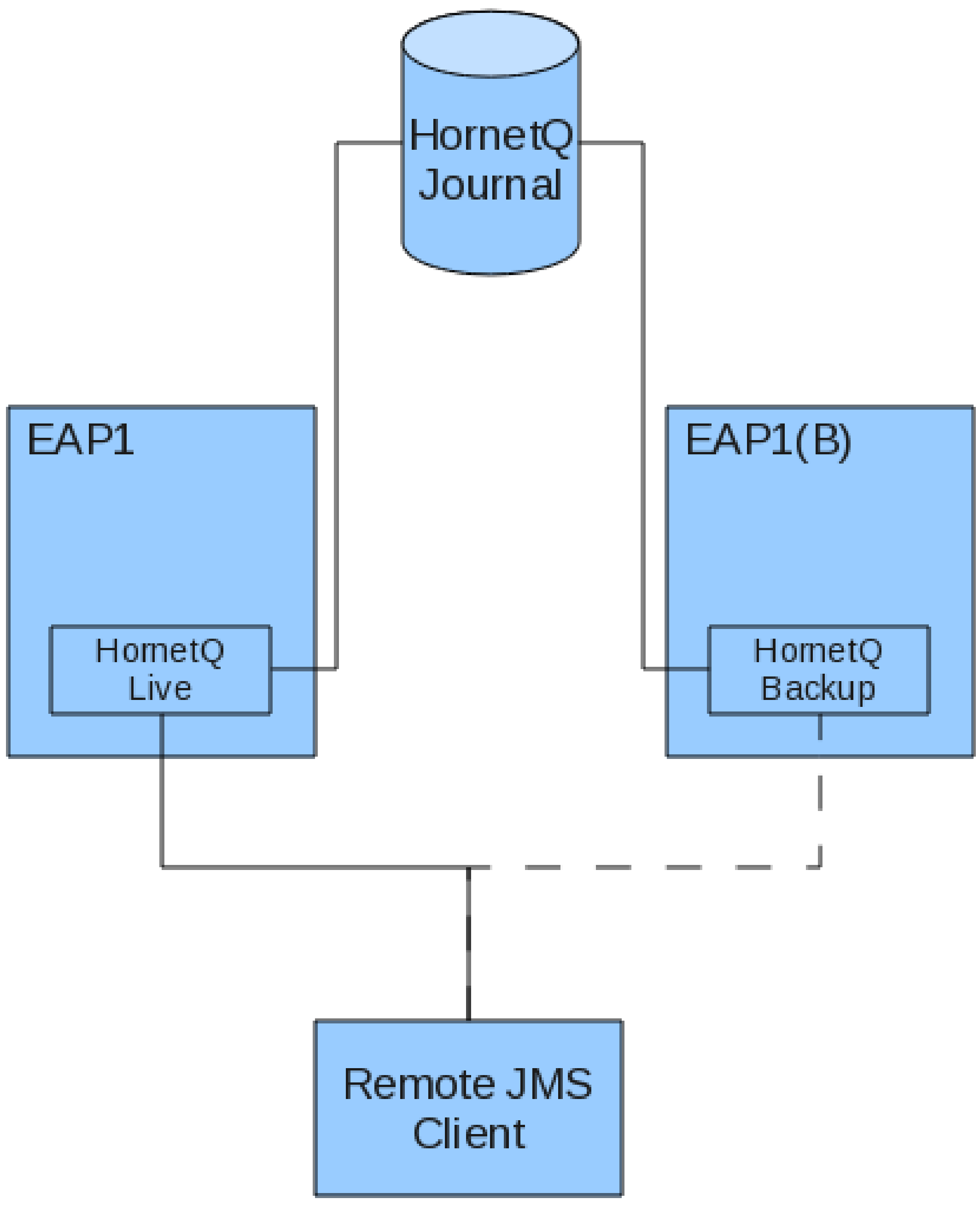
JBoss Enterprise Application Platform ships with an example configuration for this topology, located in **`$JBOSS_HOME/extras/hornetq/resources/examples/cluster-with-dedicated-backup`**.

In a dedicated symmetrical topology, the backup server resides on a separate JBoss Enterprise Application Platform instance, rather than colocated with another live server.

This means the JBoss Enterprise Application Platform instance is passive, and not used until the backup becomes live. The passive instance is therefore only useful for pure JMS applications.

The following diagram shows a possible configuration for this:

Example 38.3. Single Instance, Pure JMS, Dedicated Symmetrical Configuration



When the HornetQ live server on the EAP1 node stops responding, the HornetQ backup instance on the EAP1(B) node activates, and becomes the live server. The Remote JMS client routes all messages destined for the HornetQ live node to the HornetQ backup node.

Example 38.3, "Single Instance, Pure JMS, Dedicated Symmetrical Configuration" describes how a dedicated symmetrical topology works with applications that are pure JMS, and have no JMS components (for example, Message Driven Beans).

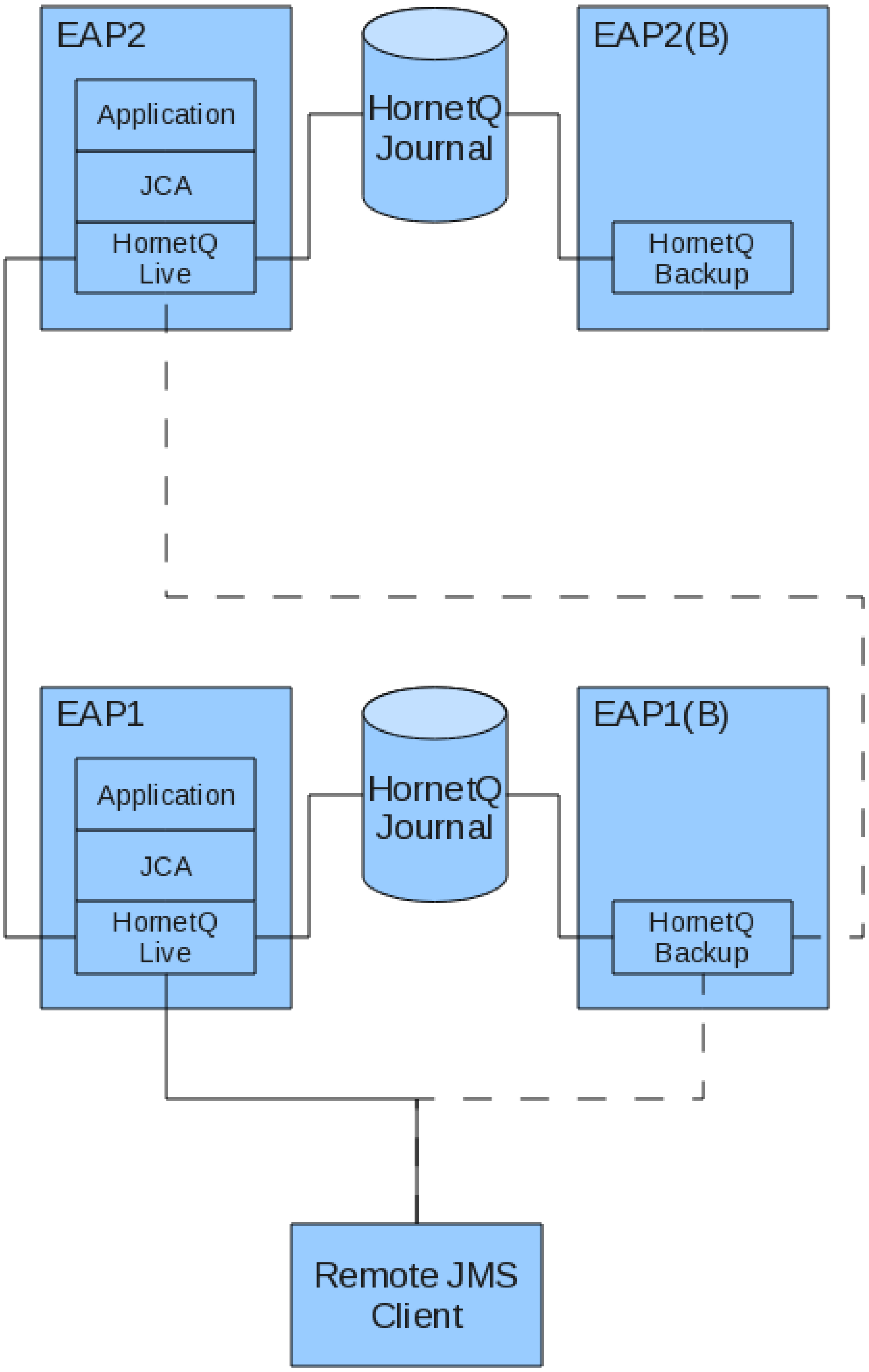
For topologies that contain JMS components, there are two approaches you can take for dedicated symmetrical clusters containing JMS components:

1. Dedicated JCA Server, as described in Example 38.4, "Dedicated JCA Server"

2. Remote JCA Server, as described in Example 38.5, "Remote JCA Server".

38.2.1. Dedicated JCA Live Server

Example 38.4. Dedicated JCA Server



The EAP1(B) and EAP2(B) instances are only running backup HornetQ instances, therefore it does not make sense to host any applications on these instances. Applications are instead hosted on EAP1 and EAP2, closely located to the live HornetQ instances.

When a live HornetQ server fails on EAP1, traffic fails over to the backup HornetQ servers EAP1(B). The remote JMS client reroutes messages sent from, or destined for Java EE components on the live server to the backup server using the HornetQ cluster connections.

The live server configuration is essentially the same as in Section 38.1.1, "Colocated Live Server". The only difference is there is no backup in the same JBoss Enterprise Application Platform node running the live HornetQ instance. This topology also requires multiple JBoss Enterprise Application Platform instances.

Procedure 38.7. Create Dedicated Live Server Profile

IMPORTANT

If nodes can not discover each other, verify that you have configured the firewall and UDP ports correctly. The network configuration should allow nodes on a cluster to communicate with each other.

You must create a copy of the **`production`** profile to customize the live server configuration.

IMPORTANT

Always copy an included profile rather than editing it directly for your custom profile. If you make a critical mistake during configuration, you can fall back to the base configuration at any time.

1. Navigate to **`$JBOSS_HOME/server/`**
2. Copy the **`production`** profile, and rename it to **`HornetQ_Dedicated`**

Procedure 38.8. Configure Shared Store and Journaling

Follow this procedure to specify HornetQ must use a shared store for fail over, and define the location of the journal files each HornetQ instance in the live backup group uses.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Add the <shared-store> element as a child of the <configuration> element.

   ```
   <shared-store>true</shared-store>
   ```

4. Ensure the bindings, journal, and large messages path locations are set to a location the live backup group can access.

   You can set absolute paths as the example describes, or use the JBoss parameters that exist in the configuration file.

If you choose the parameter option, and you do not use the default paths that these parameters resolve to, you must specify the path your bindings, journal, and large messages reside in each time you start the server.

```
<large-messages-directory>/media/shared/data/large-messages</large-
messages-directory>

<bindings-directory>/media/shared/data/bindings</bindings-directory>

<journal-directory>/media/shared/data/journal</journal-directory>

<paging-directory>/media/shared/data/paging</paging-directory>
```

NOTE

Ensure you specify paths that are accessible to the live backup groups on your network.

Procedure 38.9. Configure JMS Client Graceful Shutdown

Follow this procedure to configure how JMS clients re-establish a connection if a server is shut down gracefully.

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-configuration.xml`**
3. Specify the <fail over-on-shutdown> element in the area near the journal directory configuration in Procedure 38.2, “Configure Shared Store and Journaling”.

   ```
   <failover-on-shutdown>true</failover-on-shutdown>
   ```

   NOTE

   You are not constrained where you put the element in the **`hornetq-configuration.xml`** file, however it is easier to find the less detailed settings if they are all located at the top of the file.

4. Save and close the file.

Procedure 38.10. Configure HA Connection Factories

1. Navigate to **`$JBOSS_HOME/server/HornetQ_Dedicated/deploy/hornetq/`**
2. Open **`hornetq-jms.xml`**.
3. Add the following attributes and values as specified below.

   <ha>true</ha>

   Specifies the client must support high availability, and must always be true for fail over to occur.

<retry-interval>1000</retry-interval>

Specifies how long the client must wait (in milliseconds) before it can reconnect to the server.

<retry-interval-multiplier>1.0</retry-interval-multiplier>

Specifies the multiplier <retry-interval> uses for each subsequent reconnection pauses. By setting the value to **`1.0`**, the retry interval is the same for each client reconnection request.

<reconnect-attempts>-1</reconnect-attempts>

Specifies how many reconnect attempts a client should make before failing. Setting **`-1`** means unlimited reconnection attempts.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration xmlns="urn:hornetq"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

   <connection-factory name="NettyConnectionFactory">
      <xa>true</xa>
      <connectors>
         <connector-ref connector-name="netty"/>
      </connectors>
      <entries>
         <entry name="/ConnectionFactory"/>
         <entry name="/XAConnectionFactory"/>
      </entries>
      <ha>true</ha>

      <!-- Pause 1 second between connect attempts -->

      <retry-interval>1000</retry-interval>

      <!-- Multiply subsequent reconnect pauses by this multiplier.
This can be used
       to implement an exponential back-off. For our purposes we
just set to 1.0 so
       each reconnect pause is the same length -->

      <retry-interval-multiplier>1.0</retry-interval-multiplier>

      <!-- Try reconnecting an unlimited number of times (-1 means
unlimited) -->

      <reconnect-attempts>-1</reconnect-attempts>
   </connection-factory>

</configuration>
```

4. Define new queues in both master and backup nodes by adding one of the following configuration blocks to the specified file.

   For **`production/deploy/hornetq/hornetq-jms.xml`**

```
<queue name="testQueue">
```

```
   <entry name="/queue/testQueue"/>
   <durable>true</durable>
</queue>
```

For **production/deploy/*customName*-hornetq-jms.xml**

> **NOTE**
>
> Ensure the file is well-formed from an XML validation perspective by ensure the XML Namespace is present and correct in the file as specified.

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

   <queue name="testQueue">
      <entry name="/queue/testQueue"/>
      <durable>true</durable>
   </queue>

</configuration>
```

38.2.2. Dedicated JCA Backup Server

For the backup server, the **hornetq-configuration.xml** is unchanged. Because there is no live server, you must ensure the **hornetq-jboss-beans.xml** instantiates all the beans needed. You configure this using the same configuration as in the live server, with the exception of changing the location of the **hornetq-configuration.xml** parameter for the **Configuration** bean.

Procedure 38.11. Configure Dedicated Backup Server

Follow this procedure to configure a dedicated HornetQ backup instance, located on a separate JBoss Enterprise Application Platform server.

Prerequisites

- A HornetQ Live Server, configured according to the procedures contained within Section 38.2.1, “Dedicated JCA Live Server”
- A correctly configured JBoss Enterprise Application Platform instance, installed on a separate server instance to the live server.
- HornetQ installed on the platform instance.

1. On the backup server, navigate using the command line to **/extras/hornetq/**.
2. Execute **sh ./switch.sh -Dbackup=true**.

   The script executes and creates a **production-backup** server profile in **$JBOSS_HOME/server/**
3. Copy the **production-backup** server profile, and rename it to **HornetQ_Dedicated_Backup**.

4. Open **`$JBOSS_HOME/server/HornetQ_Dedicated_Backup/hornetq/hornetq-configuration.xml`**

5. Add the <shared-store>true</shared-store> element as a child element to the <configuration> element.

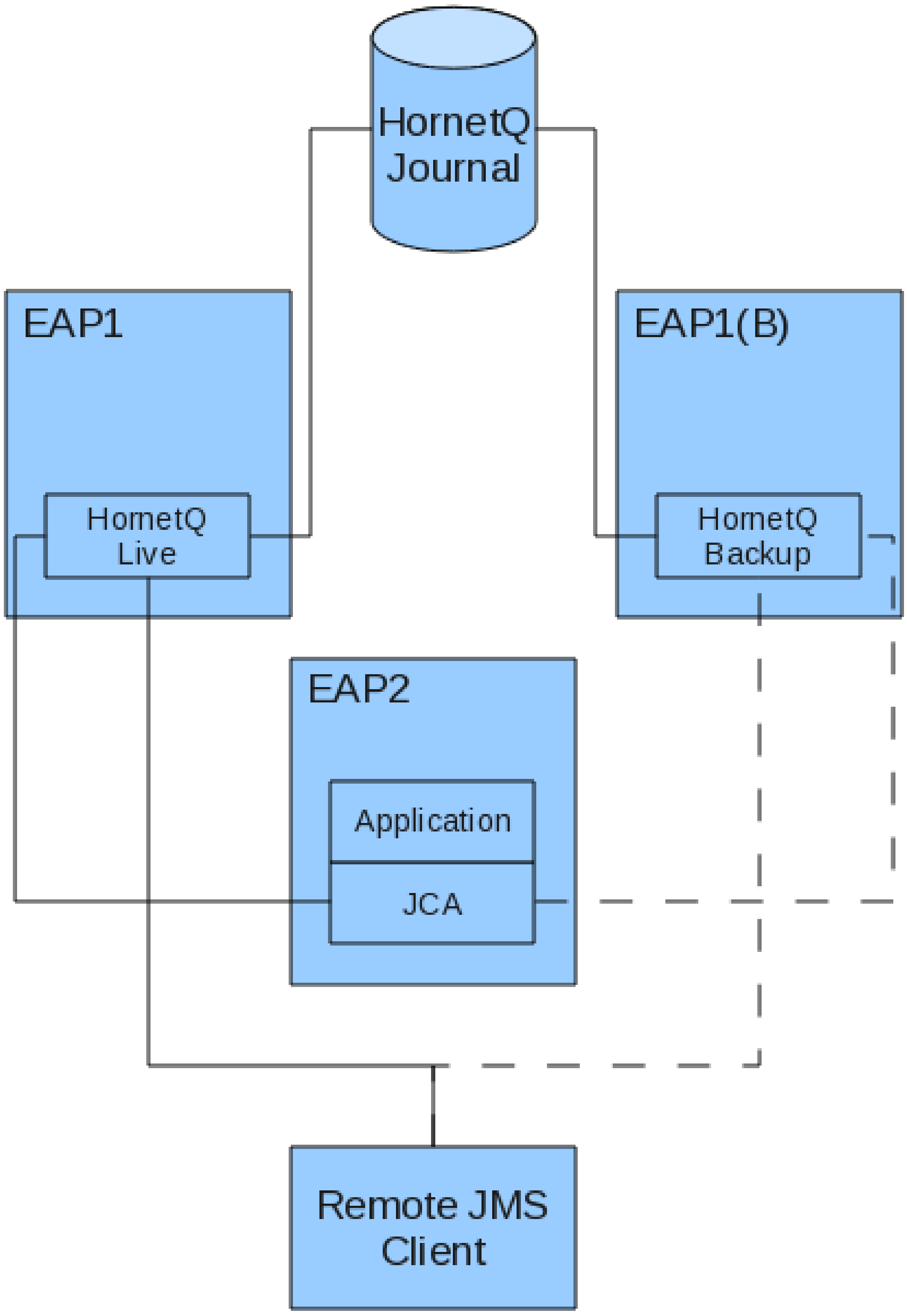
6. Change the data directory locations to match the following values:

   - <large-messages-directory>/media/shared/data/large-messages</large-messages-directory>
   - <bindings-directory>/media/shared/data/bindings</bindings-directory>
   - <journal-directory>/media/shared/data/journal</journal-directory>
   - <paging-directory>/media/shared/data/paging</paging-directory>

7. Save and close all updated files.

38.2.3. Dedicated Remote Server

Example 38.5. Remote JCA Server



In this example, the HornetQ live instance is stored on a remote server EAP1. The backup HornetQ instance is stored on EAP1(B). The JCA and the Applications are stored on a separate EAP2 instance.

When fail over occurs, the Application (via JCA) is serviced by a HornetQ server in its own JBoss Enterprise Application Platform instance.

Because both HornetQ instances are located on remote servers, you must configure the JCA connection factories on the EAP2 server to correctly serve applications to the live HornetQ server and the backup HornetQ server.

Procedure 38.12. Configure JCA Connection Factories

Follow this procedure to configure the Outbound and Inbound JCA connector elements in different configuration files. Ensure you copy the Key steps are identified by a step title.

1. Copy the `production` server profile, and rename it to `EAP2`.

2. On the EAP2 instance, navigate to `$JBOSS_HOME/server/EAP2/deploy/hornetq/jms-ds.xml`

3. The default `jms-ds.xml` has the following <config-property> configuration present in the <tx-connection-factory>.

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
   <!-- JMS Stuff -->

   <mbean code="org.jboss.jms.jndi.JMSProviderLoader"
    name="hornetq:service=JMSProviderLoader,name=JMSProvider">
      <attribute name="ProviderName">DefaultJMSProvider</attribute>
      <attribute
name="ProviderAdapterClass">org.jboss.jms.jndi.JNDIProviderAdapter</
attribute>
      <attribute
name="FactoryRef">java:/XAConnectionFactory</attribute>
      <attribute
name="QueueFactoryRef">java:/XAConnectionFactory</attribute>
      <attribute
name="TopicFactoryRef">java:/XAConnectionFactory</attribute>
   </mbean>

   <!-- JMS XA Resource adapter, use this to get transacted JMS in
beans -->

   <tx-connection-factory>
      <jndi-name>JmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-
definition>org.hornetq.ra.HornetQRAConnectionFactory</connection-
definition>
      <config-property name="SessionDefaultType"
type="java.lang.String">javax.jms.Topic</config-property>
      <config-property name="JmsProviderAdapterJNDI"
type="java.lang.String">java:/DefaultJMSProvider</config-property>
      <max-pool-size>20</max-pool-size>
      <security-domain-and-application>JmsXARealm</security-domain-
and-application>
   </tx-connection-factory>
</connection-factories>
```

4. **Configure Outbound JCA Connector**
   Add extra <config-property> elements as described in the following code sample.

   IMPORTANT

   Substitute the [live_server_IP_address] and [live_server_port_number] with the network address locations for your live server.

   If you are using Discovery to set IP address/port combinations, ensure you set the appropriate parameters for <DiscoveryAddress> and <DiscoveryPort> to match your configured broadcast groups.

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
   <!-- JMS Stuff -->

   <mbean code="org.jboss.jms.jndi.JMSProviderLoader"
    name="hornetq:service=JMSProviderLoader,name=JMSProvider">
      <attribute name="ProviderName">DefaultJMSProvider</attribute>
      <attribute
name="ProviderAdapterClass">org.jboss.jms.jndi.JNDIProviderAdapter</
attribute>
      <attribute
name="FactoryRef">java:/XAConnectionFactory</attribute>
      <attribute
name="QueueFactoryRef">java:/XAConnectionFactory</attribute>
      <attribute
name="TopicFactoryRef">java:/XAConnectionFactory</attribute>
   </mbean>

   <!-- JMS XA Resource adapter, use this to get transacted JMS in
beans -->

   <tx-connection-factory>
      <jndi-name>JmsXA</jndi-name>
      <xa-transaction/>
      <rar-name>jms-ra.rar</rar-name>
      <connection-
definition>org.hornetq.ra.HornetQRAConnectionFactory</connection-
definition>
      <config-property name="SessionDefaultType"
type="java.lang.String">javax.jms.Topic</config-property>
      <config-property name="JmsProviderAdapterJNDI"
type="java.lang.String">java:/DefaultJMSProvider</config-property>
      <config-property name="ConnectorClassName"
type="java.lang.String">org.hornetq.core.remoting.impl.netty.NettyCo
nnectorFactory</config-property>
      <config-property name="ConnectionParameters"
type="java.lang.String">host=[live_server_IP_address];port=
[live_server_port_number]</config-property>
      <max-pool-size>20</max-pool-size>
      <security-domain-and-application>JmsXARealm</security-domain-
and-application>
   </tx-connection-factory>
</connection-factories>
```

5. Open **`$JBOSS_HOME/server/EAP2/deploy/jms-ra.rar/META-INF/ra.xml`** in a text editor.

6. In **`ra.xml`**, search for <resourceadapter>.

7. **Configure the Inbound Connector**
   Replace the "The transport type" and "The transport configuration..." <config-property> elements, and their child elements with the following configuration:

   IMPORTANT

   Substitute the [live_server_IP_address] and [live_server_port_number] with the network address locations for your live server.

   If you are using Discovery to set IP address/port combinations, ensure you set the appropriate parameters for <DiscoveryAddress> and <DiscoveryPort> to match your configured broadcast groups.

   If you are using Auto Discovery, ensure you comment out ConnectorClassName and ConnectionParameters directives.

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

<!-- Preceeding parts of config file removed for readability -->

<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.netty.NettyConnectorFactory</co
nfig-property-value>
   </config-property>
   <config-property>
      <description>The transport configuration. These values must be
in the form of key=val;key=val;</description>
      <config-property-name>ConnectionParameters</config-property-
name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value>host=[live_server_IP_address];port=
[live_server_port_number]</config-property-value>
   </config-property>
   <config-property>
      <description>Do we support HA</description>
      <config-property-name>HA</config-property-name>
      <config-property-type>java.lang.Boolean</config-property-type>
      <config-property-value>true</config-property-value>
   </config-property>
```

```
<!-- Rest of config file removed for readability -->

<resourceadapter>
```

CHAPTER 39. LIBAIO NATIVE LIBRARIES

HornetQ distributes a native library, used as a bridge between HornetQ and Linux libaio.

Refer to the *Installation Guide* for instructions on downloading and installing HornetQ on Red Hat Enterprise Linux.

`libaio` is a library, developed as part of the Linux kernel project. With **`libaio`**, writes are submitted to the operating system where they are processed asynchronously. When the writes have been processed, the operating system calls the code back.

This is used in the high performance journal if configured to do so. Refer to Chapter 13, *Persistence*.

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. Refer to Section 5.1, “Library Path”.

CHAPTER 40. THREAD MANAGEMENT

This chapter describes how HornetQ uses and pools threads and how to manage them.

First we will discuss how threads are managed and used on the server side, then we will look at the client side.

40.1. 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.

When using old (blocking) IO, a separate thread pool is also used to service connections. Since old IO requires a thread per connection, it does not make sense to get them from the standard pool as the pool will easily get exhausted if too many connections are made, resulting in the server "hanging" since it has no remaining threads to do anything else. If you require the server to handle many concurrent connections, make sure to use NIO, not old IO.

When using new IO (NIO), HornetQ will, by default, use a number of threads equal to three times the number of cores (or hyper-threads) as reported by Runtime.getRuntime().availableProcessors() for processing incoming packets. To override this value, set the number of threads by specifying the parameter **`nio-remoting-threads`** in the transport configuration. Refer to Chapter 14, *Configuring the Transport* for more information on this.

There are also a small number of other places where threads are used directly:

40.1.1. 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 threads used by this pool is configured in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/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.

40.1.2. 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 threads used by this pool is configured in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`** with the **`thread-pool-max-size`** parameter.

If a value of **`-1`** is specified, the thread pool has no upper bound and new threads will be created on demand if there are not 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 **`30`**.

See the J2SE Javadoc for more information on unbounded (cached), and bounded (fixed) thread pools.

40.1.3. 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, refer to Chapter 20, *Message Expiry*.

40.1.4. 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, which avoids performance issues with context switching. You will find this thread on a thread dump with the prefix HornetQ-AIO-writer-pool.

40.2. 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, use the appropriate setter methods immediately after creation. For example:

```
ClientSessionFactory myFactory =
HornetQClient.createClientSessionFactory(...);
myFactory.setUseGlobalPools(false);
myFactory.setScheduledThreadPoolMaxSize(10);
myFactory.setThreadPoolMaxSize(-1);
```

When using the JMS API, set the same parameters on the ClientSessionFactory and use it to create the **`ConnectionFactory`** instance. For example:

```
ConnectionFactory myConnectionFactory =
HornetQJMSClient.createConnectionFactory(myFactory);
```

If you are using JNDI to instantiate **`HornetQConnectionFactory`** instances, you can also set these parameters in the ***`JBOSS_DIST`*`/jboss-as/server/`*`<PROFILE>`*`/deploy/hornetq/hornetq-jms.xml`** file where you describe your connection factory. For example:

```
<connection-factory name="NettyConnectionFactory">
```

```
   <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>
```

CHAPTER 41. LOGGING

By default, HornetQ uses log4j to manage logging events within JBoss Enterprise Application Platform. The default delegate sends all log requests to the standard JDK logging, also known as Java-Util-Logging (JUL).

NOTE

For detailed information about configuring JUL visit the Java Logging Overview page on the Oracle website.

HornetQ does have its own logging delegate that has no dependencies on any particular logging framework.

By default the server picks up its JUL configuration from a **`logging.properties`** file found in the config directories. This is configured to use the HornetQ logging formatter (**`HornetQLoggerFormatter.java`**) and will log to the console as well as a log file.

You can configure a different Logging Delegate programmatically or via a System Property.

To do this programmatically, do the following where **`Log4jLogDelegateFactory`** is the implementation of **`org.hornetq.spi.core.logging.LogDelegateFactory`** that you would like to use:

```
org.hornetq.core.logging.Logger.setDelegateFactory(new
Log4jLogDelegateFactory())
```

To do this via a System Property, set the property **`org.hornetq.logger-delegate-factory-class-name`** to the delegate factory being used. For example:

```
-Dorg.hornetq.logger-delegate-factory-class-
name=org.hornetq.integration.logging.Log4jLogDelegateFactory
```

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

To configure the client's logging to use the JUL delegate, ensure that a **`logging.properties`** file is provided and set the java.util.logging.config.file property on client start up.

CHAPTER 42. INTERCEPTING OPERATIONS

HornetQ supports *interceptors* to intercept packets entering the server. Any supplied interceptors are called for any packet entering the server. This allows custom code to be executed, such as for auditing packets or filtering. Interceptors can change the packets they intercept.

42.1. IMPLEMENTING THE INTERCEPTORS

An 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.

42.2. CONFIGURING THE INTERCEPTORS

The interceptors are configured in **`<`*`JBOSS_HOME`*`>/jboss-as/server/<`*`PROFILE`*`>/deploy/hornetq/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.

42.3. 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.

CHAPTER 43. PERFORMANCE TUNING

In this chapter we will discuss how to tune HornetQ for optimum performance.

43.1. TUNING PERSISTENCE

- Put the message journal on its own physical volume. For example, if the disk is shared with a transaction coordinator, database, or other journals which are also reading and writing from the disk, these may greatly reduce performance because the disk head may be skipping between the different files. One advantage of an append only journal is that disk head movement is minimized - this advantage is destroyed if the disk is shared. If using paging or large messages, ideally, ensure that they are put on separate volumes.
- Minimum number of journal files. Set **`journal-min-files`** to a number of files that would fit the average sustainable rate. For example, if new files are being created on the journal data directory too often and lots of data is being persisted, increase the minimum 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 the 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 running AIO, it may be possible to increase performance by increasing **`journal-max-io`**. DO NOT change this parameter if running NIO.

43.2. TUNING JMS

Optimization is possible in the following areas when using the JMS API:

- Disable message id. Use the **`setDisableMessageID()`** method on the **`MessageProducer`** class to disable message ids if not needed. This decreases the size of the message and also avoids the overhead of creating a unique ID.
- Disable message time stamp. Use the **`setDisableMessageTimeStamp()`** method on the **`MessageProducer`** class to disable message timestamps not required.
- 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 verbose, resulting in increased server traffic, also Java serialization is slow in comparison to custom marshaling techniques. Only use **`ObjectMessage`** if one of the other message types are unsuitable (for example, if the type of payload is unknown until run-time).
- Avoid **`AUTO_ACKNOWLEDGE`**. **`AUTO_ACKNOWLEDGE`** mode requires an acknowledgment to be sent from the server for each message received on the client, this means more traffic on the network. If possible, use **`DUPS_OK_ACKNOWLEDGE`**, or use **`CLIENT_ACKNOWLEDGE`** or a transacted session and batch up many acknowledgments with one acknowledge/commit.
- Avoid durable messages. By default JMS messages are durable. If really durable messages are not required, set the messages to be non-durable. Durable messages incur much more overhead in persisting them to storage.

- Batch many sends or acknowledgments in a single transaction. HornetQ will only require a network round trip on the commit, not on every send or acknowledgment.

43.3. OTHER TUNINGS

There are various other places in HornetQ where tuning can be performed:

- Use Asynchronous Send Acknowledgments. To send durable messages non transactionally and a guarantee is required that they have reached the server by the time the call to send() returns, do not set durable messages to be sent blocking, rather use asynchronous send acknowledgments to get the acknowledgments of send back in a separate stream. Refer to Chapter 18, *Guarantees of sends and commits* for more information on this.

- Use pre-acknowledge mode. With pre-acknowledge mode, messages are acknowledged **`before`** being sent to the client. This reduces the amount of acknowledgment traffic being transmitted. For more information on this, refer to Chapter 27, *Pre-Acknowledge Mode*.

- Disable persistence. If message persistence is not required, turn it off altogether by setting **`persistence-enabled`** to false in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**.

- Sync transactions lazily. Setting **`journal-sync-transactional`** to **`false`** in **`hornetq-configuration.xml`** gives better transactional persistent performance at the expense of some possibility of loss of transactions on failure. Refer to Chapter 18, *Guarantees of sends and commits* for more information.

- Sync non transactional lazily. Setting **`journal-sync-non-transactional`** to **`false`** in **`hornetq-configuration.xml`** can provide better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure. Refer to Chapter 18, *Guarantees of sends and commits* for more information.

- Send messages non blocking. Setting **`block-on-durable-send`** and **`block-on-non-durable-send`** to **`false`** in **`JBOSS_DIST/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-jms.xml`** (if using JMS and JNDI) or directly on the ClientSessionFactory. It is therefore not required to wait an entire network round trip for every message sent. Refer to Chapter 18, *Guarantees of sends and commits* for more information.

- For very fast consumers, increase consumer-window-size. This effectively disables consumer flow control.

- Socket NIO vs Socket Old IO. By default HornetQ uses old (blocking) on the server and the client side (Refer to Chapter 14, *Configuring the Transport* for more information). NIO is much more scalable but can give some latency hit compared to old blocking IO. To be able to service many thousands of connections on the server, then ensure the use of NIO on the server. However, for fewer connections on the server, retaining the old IO for the server acceptors may gain a small performance advantage.

- Use the core API not JMS. Using the JMS API 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 transmitted, so if you re-use **`SimpleString`** instances between calls, some unnecessary copying can be avoided.

43.4. TUNING TRANSPORT SETTINGS

- TCP buffer sizes. Fast networks and fast machines may get a performance boost by increasing the TCP send and receive buffer sizes. Refer to Chapter 14, *Configuring the Transport* for more information.

  NOTE

  Note that some operating systems like later versions of Linux include TCP auto-tuning and setting TCP buffer sizes manually can prevent auto-tune from working and actually give you worse performance!

- Increase limit on file handles on the server. If a lot of concurrent connections on the servers is expected, or if clients are rapidly opening and closing connections, ensure that the user running the server has permission to create sufficient file handles.

  This varies from operating system to operating system. On Linux systems, increase the number of allowable open file handles in the file **/etc/security/limits.conf**, for example. add the lines

  ```
  serveruser     soft    nofile  20000
  serveruser     hard    nofile  20000
  ```

  This would allow up to 20 000 file handles to be open by the user **serveruser**.

- Use **batch-delay** and set **direct-deliver** to false for the best throughput for very small messages. HornetQ comes with a pre-configured connector/acceptor pair (**netty-throughput**) in**<*JBOSS_HOME*>/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-configuration.xml** and JMS connection factory (**ThroughputConnectionFactory**) in ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml** which can be used to give the very best throughput, especially for small messages. Refer to Chapter 14, *Configuring the Transport* for more information.

43.5. AVOIDING ANTI-PATTERNS

- Re-use connections, sessions, consumers, and producers. Probably the most common messaging anti-pattern observed, is users who create a new connection, session, or producer for every message sent, or every message consumed. This is a poor use of resources. Always reuse these objects as they take time to create and may involve several network round trips.

  NOTE

  Some popular libraries such as the Spring JMS Template are known to use these anti-patterns. If using Spring JMS Template, it is possibly a cause for poor performance and not HornetQ. The Spring JMS Template can only safely be used in an app server which caches JMS sessions (for example. using JCA), and only then for sending messages. It cannot be safely be used for synchronously consuming messages, even in an app server.

- Avoid fat messages. Verbose formats such as XML result in increased server transmission load and performance will suffer as result. Avoid XML in message bodies if possible.

- Do not 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.

- Do not use Message-Driven Beans unnecessarily. MDB usage greatly increases the code path for each message received compared to a straightforward message consumer, as a lot of extra application server code is executed. Before using MDBs, investigate the use of a normal message consumer to complete the task.

APPENDIX A. CONFIGURATION REFERENCE

This section is a quick index for looking up configuration values. Click on the element name to go to the specific chapter.

A.1. SERVER CONFIGURATION

A.1.1. hornetq-configuration.xml

The **`hornetq-configuration.xml`** file contains the core server configuration. The file is located in **`<JBOSS_HOME>/jboss-as/server/<PROFILE>/deploy/hornetq/hornetq-configuration.xml`**.

Table A.1. Server Configuration

Element Name	Type	Description	Default
backup	Boolean	if **`true`**, this server is a backup to another node in the cluster.	false
allow-failback	Boolean	Specifies whether the backup server automatically stops and returns to standby mode when the live server becomes available again. If set to **`false`**, the server must be stopped manually to trigger a return to standby mode.	true
failover-on-shutdown	Boolean	Specifies how fail-over behaves when a live server is shutdown correctly. If set to **`true`**, the backup HornetQ instance takes over when the live server is shut down gracefully.	false
shared-store		Specifies whether the server should reference a shared store for journaling.	false
grouping-handler	Parent element	Used to specify the LOCAL and REMOTE grouping handlers.	
remoting-interceptors	Parent element	Used to specify class names, using <class-name>. Refer to Chapter 42, *Intercepting Operations* for detailed information about this element.	
address-full-policy	String	Supports three values: PAGE, DROP, and BLOCK. Refer to Section 22.3, “Paging Mode”	PAGE

Element Name	Type	Description	Default
send-to-dla-on-no-route	Boolean	Specifies how messages are handled when a server does not routes the message to a queue. If set to `true`, the message is sent to the dead letter address (DLA) for the routed address, if a DLA exists. If a DLA does not exist, the message is dropped as a last resort. Refer to Section 23.3, "Configuring Queues Through Address Settings" for more information about Address Setting elements.	false
backup-connector-ref	String	the name of the remoting connector to connect to the backup node.	
bindings-directory	String	the directory in which to store the persisted bindings.	data/bindings
clustered	Boolean	if `true`, the server is clustered.	false
connection-ttl-override	Long	if set, overrides the time (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	if `true`, the `journal` directory will be created.	true
file-deployment-enabled	Boolean	if `true`, the server loads configuration from the configuration files.	true
id-cache-size	Integer	the size of the cache for pre-creating message IDs.	2000
journal-buffer-size	Long	The size of the internal buffer on the journal.	128 kilobytes
journal-buffer-timeout	Long	the timeout (in nanoseconds) used to flush internal buffers on the journal.	20000

Element Name	Type	Description	Default
journal-compact-min-files	Integer	the minimum number of data files before compacting occurs.	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	the number of 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	String	the type of journal to use.	NIO
jmx-management-enabled	Boolean	if **`true`**, the management API is available via JMX.	true
jmx-domain	String	the JMX domain used to registered HornetQ MBeans in the MBeanServer.	org.hornetq
large-messages-directory	String	the directory in which to store large messages. The default location is **`data/largemessages`**	

Element Name	Type	Description	Default
management-address	String	the name of the management address to send management messages to. The default value is **`jms.queue.hornetq.management`**	
cluster-user	String	the user used by cluster connections to communicate between the clustered nodes. The default value is **`HORNETQ.CLUSTER.ADMIN.USER`**	See Description
cluster-password	String	the password used by cluster connections to communicate between the clustered nodes.	CHANGE ME!!
management-notification-address	String	the name of the address that consumers bind to receive management notifications. The default value is **`hornetq.notifications`**	See Description
message-counter-enabled	Boolean	if **`true`**, 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
page-max-concurrent-io	Integer	The maximum number of concurrent reads the system can make on the paging files.	5
persist-delivery-count-before-delivery	Boolean	if **`true`**, delivery count is persisted before delivery; if **`false`**, this occurs only after a message has been canceled.	false

Element Name	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	if `true`, 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` sets unlimited threads.	30
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
connectors	String	a list of remoting connectors configurations to create.	
connector.name (attribute)	String	Name of the connector - mandatory.	

Element Name	Type	Description	Default
connector.factory-class	String	Name of the ConnectorFactory implementation - mandatory.	
connector.param	String	A key-value pair used to configure the connector. A connector can have many param.	
connector.param.key (attribute)	String	Key of a configuration parameter - mandatory.	
connector.param.value (attribute)	String	Value of a configuration parameter - mandatory.	
acceptors	String	a list of remoting acceptors to create.	
acceptor.name (attribute)	String	Name of the acceptor - optional.	
acceptor.factory-class	String	Name of the AcceptorFactory implementation - mandatory.	
acceptor.param	String	A key-value pair used to configure the acceptor. An acceptor can have many param.	
acceptor.param.key (attribute)	String	Key of a configuration parameter - mandatory.	
acceptor.param.value (attribute)	String	Value of a configuration parameter - mandatory.	
broadcast-groups		a list of broadcast groups to create.	
broadcast-group.name (attribute)	String	a unique name for the broadcast group - mandatory.	
broadcast-group.local-bind-address	String	local bind address that the datagram socket is bound to. The default value is the wildcard IP address chosen by the kernel	See Description
broadcast-group.local-bind-port	Integer	local port to which the datagram socket is bound to.	-1 (anonymous port)
broadcast-group.group-address	String	multicast address to which the data will be broadcast - mandatory.	

Element Name	Type	Description	Default
broadcast-group.group-port	Integer	UDP port number used for broadcasting - mandatory.	
broadcast-group.broadcast-period	Long	period in milliseconds between consecutive broadcasts.	2000 (ms)
broadcast-group.connector-ref	Integer	A pair connector and optional backup connector that will be broadcast. A broadcast-group can have multiple connector-ref.	
broadcast-group.connector-ref.connector-name (attribute)	String	Name of the live connector - mandatory.	
broadcast-group.connector-ref.backup-connector-name (attribute)	String	Name of the backup connector - optional.	
discovery-groups	String	a list of discovery groups to create.	
discovery-group.name (attribute)	String	A unique name for the discovery group - mandatory.	
discovery-group.local-bind-address	String	The discovery group will be bound only to this local address.	
discovery-group.group-address	String	Multicast IP address of the group to listen on - mandatory.	
discovery-group.group-port	Integer	UDP port of the multicast group - mandatory	
discovery-group.refresh-timeout	Integer	Period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list.	5000 (ms)
diverts	String	A list of diverts to use.	
divert.name (attribute)	String	A unique name for the divert - mandatory.	

Element Name	Type	Description	Default
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	String	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	String	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 - 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)

Element Name	Type	Description	Default
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.fail-over-on-server-shutdown	Boolean	Should fail-over 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	String	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
cluster-connections.min-large-message-size	Integer	Message size threshold over which the message will be split into multiple packages when sent over the cluster.	100 kB
cluster-connections.reconnect-attempts	Integer	Number of times the system will try to connect a node on the cluster, after which (if max-retry has been reached) the node will be considered permanently down and the system will stop routing messages to this node.	-1 (infinite retries)

Element Name	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
cluster-connections.min-large-message-size	Integer	Maximum threshold of message size, over which it will be split into multiple packages when sent over the cluster.	100 kB
cluster-connections.reconnect-attempts	Integer	Maximum number of times the system will try to connect a node on the cluster. If the max-retry is achieved this node will be considered permanently down and the system will stop routing messages to this node.	-1 (infinite retries)
security-settings	String	A list of security settings.	
security-settings.match (attribute)	String	The string to use for matching security against an address.	
security-settings.permission	String	A permission to add to the address.	
security-settings.permission.type (attribute)	String	The type of permission.	
security-settings.permission.roles (attribute)	String	A comma-separated list of roles to apply the permission to.	
address-settings	String	A list of address settings.	

Element Name	Type	Description	Default
address-settings.dead-letter-address	String	The address to send dead messages to.	
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 to.	
address-settings.redelivery-delay	Long	The time (in ms) to wait before redelivering a canceled message.	60000 (ms)
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.	60000 (1 minute)
initial-wait-timeout	String		
server-dump-interval	Long		
connector-services.connector-service			
bridges.user			
bridges.password			
bridges.static-connectors			
cluster-connections.static.connectors			
message-counter-history-day-limit			

A.1.2. `hornetq-configuration.xsd` Reference

Below is the **`hornetq-configuration.xsd`** file that governs the validity of the **`hornetq-configuration.xml`** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="urn:hornetq"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="urn:hornetq" version="1.0">

    <xsd:element name="configuration">
        <xsd:complexType>
            <xsd:all>
                <xsd:element maxOccurs="1" minOccurs="0" name="name"
type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
ref="clustered"/>
                <xsd:element maxOccurs="1" minOccurs="0" ref="file-
deployment-enabled"/>
                <xsd:element maxOccurs="1" minOccurs="0"
ref="persistence-enabled"/>
                <xsd:element maxOccurs="1" minOccurs="0" name="scheduled-
thread-pool-max-size" type="xsd:int">
                    <xsd:annotation>
                        <xsd:documentation>
                            Maximum number of threads to use for the
scheduled thread pool
                        </xsd:documentation>
                    </xsd:annotation>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="thread-
pool-max-size" type="xsd:int">
                    <xsd:annotation>
                        <xsd:documentation>
                            Maximum number of threads to use for the
thread pool
                        </xsd:documentation>
                    </xsd:annotation>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="security-
enabled" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="security-
invalidation-interval" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="wild-card-
routing-enabled" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="management-address" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="management-notification-address" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="cluster-
user" type="xsd:string">
```

```
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="cluster-
password" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="log-
delegate-factory-class-name" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="jmx-
management-enabled" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="jmx-
domain" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="message-
counter-enabled" type="xsd:boolean">
                </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="message-
counter-sample-period" type="xsd:long">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="message-
counter-max-day-history" type="xsd:int">
            </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="connection-ttl-override" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="async-
connection-execution-enabled" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="transaction-timeout" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="transaction-timeout-scan-period" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="message-
expiry-scan-period" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="message-
expiry-thread-priority" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="id-cache-
size" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="persist-
id-cache" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" ref="remoting-
interceptors">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="backup"
type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="allow-
failback" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="failback-
```

```
delay" type="xsd:long">
                </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="failover-on-
shutdown" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="shared-
store" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="persist-
delivery-count-before-delivery" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="live-
connector-ref" type="live-connectorType">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="connectors">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="connector" type="connectorType"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0"
name="acceptors">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="1" name="acceptor" type="acceptorType">
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="broadcast-
groups">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" ref="broadcast-group">
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="discovery-
groups">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" ref="discovery-group">
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="diverts">
                    <xsd:complexType>
                        <xsd:sequence>
```

```
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="divert" type="divertType">
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="queues">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="bridges">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="bridge" type="bridgeType">
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="cluster-
connections">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="cluster-connection" type="clusterConnectionType">
                        </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="grouping-
handler" type="groupingHandlerType">
            </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="paging-
directory" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="bindings-
directory" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="create-
bindings-dir" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
directory" type="xsd:string">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="create-
journal-dir" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
type" type="journalType">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
buffer-timeout" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
buffer-size" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
sync-transactional" type="xsd:boolean">
```

```
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
sync-non-transactional" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="log-
journal-write-rate" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
file-size" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
min-files" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
compact-percentage" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
compact-min-files" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="journal-
max-io" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="perf-
blast-pages" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="run-sync-
speed-test" type="xsd:boolean">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="server-
dump-interval" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="memory-
warning-threshold" type="xsd:int">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="memory-
measure-interval" type="xsd:long">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="large-
messages-directory" type="xsd:string">
                </xsd:element>
             <xsd:element maxOccurs="1" minOccurs="0" name="security-
settings">
             </xsd:element>
             <xsd:element maxOccurs="1" minOccurs="0" name="address-
settings">
             </xsd:element>
             <xsd:element maxOccurs="1" minOccurs="0" name="connector-
services">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="connector-service" type="connectorServiceType"/>
                        </xsd:sequence>
                    </xsd:complexType>
              </xsd:element>
            </xsd:all>
```

```
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="clustered" type="xsd:boolean"/>

    <xsd:element name="file-deployment-enabled" type="xsd:boolean"/>

    <xsd:element name="persistence-enabled" type="xsd:boolean"/>

    <xsd:element name="local-bind-address" type="xsd:string"/>

    <xsd:element name="local-bind-port" type="xsd:int"/>

    <xsd:element name="group-address" type="xsd:string"/>

    <xsd:element name="group-port" type="xsd:int"/>

    <xsd:element name="broadcast-period" type="xsd:long"/>

    <xsd:element name="broadcast-group">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="1" minOccurs="0" ref="local-bind-
address">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" ref="local-bind-
port">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="1" ref="group-
address">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="1" ref="group-
port">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" ref="broadcast-
period">
                </xsd:element>
                <xsd:element maxOccurs="unbounded" minOccurs="0"
name="connector-ref" type="xsd:string">
                </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="refresh-timeout" type="xsd:int"/>

    <xsd:element name="initial-wait-timeout" type="xsd:int"/>

    <xsd:element name="discovery-group">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="1" minOccurs="0" ref="local-bind-
address">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="1" ref="group-
```

```
address">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="1" ref="group-
port">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" ref="refresh-
timeout">
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" ref="initial-
wait-timeout">
                </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="discovery-group-ref">
        <xsd:complexType>
            <xsd:attribute name="discovery-group-name" type="xsd:IDREF">
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="remoting-interceptors">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="1"
name="class-name" type="xsd:string">
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="paramType">
        <xsd:attribute name="key" type="xsd:string" use="required"/>
        <xsd:attribute name="value" type="xsd:string" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="connectorType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="factory-class"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="param"
type="paramType">
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:ID" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="acceptorType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="factory-class"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="param"
```

```
type="paramType">
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="optional"/>
    </xsd:complexType>

    <xsd:complexType name="bridgeType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="queue-name"
type="xsd:IDREF">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="forwarding-
address" type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="ha"
type="xsd:boolean">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="filter">
                <xsd:complexType>
                   <xsd:attribute name="string" type="xsd:string"
use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="transformer-
class-name" type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="retry-
interval" type="xsd:long">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="retry-
interval-multiplier" type="xsd:double">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="reconnect-
attempts" type="xsd:int">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="failover-on-
server-shutdown" type="xsd:boolean">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="use-duplicate-
detection" type="xsd:boolean">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="confirmation-
window-size" type="xsd:int">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="user"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="password"
type="xsd:string">
            </xsd:element>
            <xsd:choice>
                <xsd:element maxOccurs="1" minOccurs="1" name="static-
connectors">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
```

```
minOccurs="1" name="connector-ref" type="xsd:string"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="1" name="discovery-
group-ref">
                    <xsd:complexType>
                        <xsd:attribute name="discovery-group-name"
type="xsd:IDREF" use="required">
                        </xsd:attribute>
                    </xsd:complexType>
                </xsd:element>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>

   <xsd:complexType name="clusterConnectionType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="address"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="1" name="connector-ref"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="retry-
interval" type="xsd:long">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="use-duplicate-
detection" type="xsd:boolean">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="forward-when-
no-consumers" type="xsd:boolean">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="max-hops"
type="xsd:int">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="confirmation-
window-size" type="xsd:int">
            </xsd:element>
            <xsd:choice>
                <xsd:element maxOccurs="1" minOccurs="0" name="static-
connectors">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element maxOccurs="unbounded"
minOccurs="0" name="connector-ref" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="allow-direct-connections-
only" type="xsd:boolean" use="optional"/>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element maxOccurs="1" minOccurs="0" name="discovery-
group-ref">
                    <xsd:complexType>
                        <xsd:attribute name="discovery-group-name"
```

```
type="xsd:IDREF" use="required">
                        </xsd:attribute>
                    </xsd:complexType>
                </xsd:element>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="divertType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="0" name="routing-name"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="1" name="address"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="1" name="forwarding-
address" type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="filter">
                <xsd:complexType>
                   <xsd:attribute name="string" type="xsd:string"
use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="transformer-
class-name" type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="exclusive"
type="xsd:boolean">
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>

    <xsd:simpleType name="journalType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="ASYNCIO"/>
            <xsd:enumeration value="NIO"/>
        </xsd:restriction>
    </xsd:simpleType>

   <xsd:complexType name="groupingHandlerType">
      <xsd:sequence>
         <xsd:element maxOccurs="1" minOccurs="1" name="type"
type="groupingHandlerTypeType"/>
         <xsd:element maxOccurs="1" minOccurs="1" name="address"
type="xsd:string"/>
         <xsd:element maxOccurs="1" minOccurs="0" name="timeout"
type="xsd:int"/>
      </xsd:sequence>
       <xsd:attribute name="name" type="xsd:string" use="required"/>
   </xsd:complexType>

   <xsd:simpleType name="groupingHandlerTypeType">
```

```
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="LOCAL"/>
            <xsd:enumeration value="REMOTE"/>
         </xsd:restriction>
      </xsd:simpleType>

     <xsd:element name="security-settings">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element maxOccurs="unbounded" minOccurs="0"
name="security-setting">
               </xsd:element>
             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>

      <xsd:element name="security-setting">
       <xsd:complexType>
           <xsd:sequence>
              <xsd:element maxOccurs="unbounded" minOccurs="0"
name="permission">
                 <xsd:complexType>
                    <xsd:attribute name="type" type="xsd:string"
use="required"/>
                    <xsd:attribute name="roles" type="xsd:string"
use="required"/>
                 </xsd:complexType>
              </xsd:element>
           </xsd:sequence>
               <xsd:attribute name="match" type="xsd:string"
use="required"/>
           </xsd:complexType>
      </xsd:element>

     <xsd:element name="address-settings">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element maxOccurs="unbounded" minOccurs="0"
name="address-setting">
               </xsd:element>
             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>

      <xsd:element name="address-setting">
       <xsd:complexType>
         <xsd:all>
           <xsd:element maxOccurs="1" minOccurs="0" name="dead-letter-
address" type="xsd:string">
           </xsd:element>
           <xsd:element maxOccurs="1" minOccurs="0" name="expiry-address"
type="xsd:string">
           </xsd:element>
           <xsd:element maxOccurs="1" minOccurs="0" name="redelivery-delay"
type="xsd:long">
           </xsd:element>
```

```
        <xsd:element maxOccurs="1" minOccurs="0" name="max-delivery-
attempts" type="xsd:int">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="max-size-bytes"
type="xsd:long">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="page-size-bytes"
type="xsd:long">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="page-max-cache-
size" type="xsd:int">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="address-full-
policy" type="addressFullMessagePolicyType">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="message-counter-
history-day-limit" type="xsd:int">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="last-value-queue"
type="xsd:boolean">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="redistribution-
delay" type="xsd:long">
        </xsd:element>
        <xsd:element maxOccurs="1" minOccurs="0" name="send-to-dla-on-no-
route" type="xsd:boolean">
        </xsd:element>
      </xsd:all>
     <xsd:attribute name="match" type="xsd:string" use="required"/>
   </xsd:complexType>
   </xsd:element>

   <xsd:element name="queues">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
name="queue">
            </xsd:element>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="queue">
      <xsd:complexType>
        <xsd:all>
            <xsd:element maxOccurs="1" minOccurs="1" name="address"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="filter">
                <xsd:complexType>
                   <xsd:attribute name="string" type="xsd:string"
use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element maxOccurs="1" minOccurs="0" name="durable"
type="xsd:boolean">
```

```
            </xsd:element>
        </xsd:all>
        <xsd:attribute name="name" type="xsd:ID" use="required"/>
       </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="live-connectorType">
        <xsd:attribute name="connector-name" type="xsd:IDREF"
use="required">
        </xsd:attribute>
    </xsd:complexType>

    <xsd:simpleType name="addressFullMessagePolicyType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="DROP"/>
            <xsd:enumeration value="PAGE"/>
            <xsd:enumeration value="BLOCK"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="connectorServiceType">
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="factory-class"
type="xsd:string">
            </xsd:element>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="param"
type="paramType">
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="optional"/>
    </xsd:complexType>

</xsd:schema>
```

A.1.3. An Example `hornetq-configuration.xml`

Below is an example configuration file:

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

   <security-settings>
      <security-setting match="jms.queue.testQueue">
         <permission type="consume" roles="guest,publisher"/>
         <permission type="send" roles="guest,publisher"/>
      </security-setting>
   </security-settings>

   <address-settings>
      <!--default for catch all-->
      <address-setting match="#">
         <dead-letter-address>jms.queue.DLQ</dead-letter-address>
         <expiry-address>jms.queue.ExpiryQueue</expiry-address>
         <redelivery-delay>0</redelivery-delay>
         <max-size-bytes>-1</max-size-bytes>
         <page-size-bytes>10485760</page-size-bytes>
         <message-counter-history-day-limit>10</message-counter-history-
day-limit>
      </address-setting>
   </address-settings>

</configuration>
```

A.1.4. hornetq-jms.xml

This is the configuration file used by the server side JMS service to load JMS Queues, Topics and Connection Factories. It is located by default in ***JBOSS_DIST*/jboss-as/server/<*PROFILE*>/deploy/hornetq/hornetq-jms.xml**.

Table A.2. JMS Server Configuration

Element Name	Type	Description	Default
connection-factory		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 by 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.discovery-group-name (attribute)	String	Name of discovery group used by this connection factory.	
connection-factory.discovery-initial-wait-timeout	Long	The initial time to wait (in ms) for discovery groups to wait for broadcasts.	10000
connection-factory.block-on-acknowledge	Boolean	Whether or not messages are acknowledged synchronously.	false

Element Name	Type	Description	Default
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. The default is: **`org.hornetq.api.core.client.loadbalance.roundRobinConnectionLoadBalancingPolicy`**	See description
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 acknowledgments when using **`DUPS_OK_ACKNOWLEDGE`** mode.	1024 * 1024
connection-factory.fail-over-on-initial-connection	Boolean	Whether or not to fail-over to backup on event that initial connection to live server fails.	false
connection-factory.fail-over-on-server-shutdown	Boolean	Whether or not to fail-over on server shutdown.	false

Element Name	Type	Description	Default
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.	1.0
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-factory.transaction-batch-size	Integer	The batch size (in bytes) between acknowledgments when using a transactional session.	1024 * 1024
connection-factory.use-global-pools	Boolean	Whether or not to use a global thread pool for threads.	true

Element Name	Type	Description	Default
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).	

APPENDIX B. RA.XML HORNETQ RESOURCE ADAPTER FILE

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

<!-- $Id: ra.xml 76819 2008-08-08 11:04:20Z jesper.pedersen $ -->

<connector 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/connector_1_5.xsd"
           version="1.5">

   <description>HornetQ 2.0 Resource Adapter</description>
   <display-name>HornetQ 2.0 Resource Adapter</display-name>

   <vendor-name>Red Hat Middleware LLC</vendor-name>
   <eis-type>JMS 1.1 Server</eis-type>
   <resourceadapter-version>1.0</resourceadapter-version>

   <license>
      <description>
Copyright 2009 Red Hat, Inc.
 Red Hat licenses this file to you under the Apache License, version
 2.0 (the "License"); you may not use this file except in compliance
 with the License.  You may obtain a copy of the License at
   http://www.apache.org/licenses/LICENSE-2.0
 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 implied.  See the License for the specific language governing
 permissions and limitations under the License.
      </description>
      <license-required>true</license-required>
   </license>

   <resourceadapter>
      <resourceadapter-
class>org.hornetq.ra.HornetQResourceAdapter</resourceadapter-class>
      <config-property>
         <description>
            The transport type. Multiple connectors can be configured by
using a comma separated list,
            i.e.
org.hornetq.core.remoting.impl.invm.InVMConnectorFactory,org.hornetq.core.
remoting.impl.invm.InVMConnectorFactory.
         </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.InVMConnectorFactory</config-
property-value>
      </config-property>
      <config-property>
         <description>The transport configuration. These values must be
in the form of key=val;key=val;,
            if multiple connectors are used then each set must be
```

```
separated by a comma i.e. host=host1;port=5445,host=host2;port=5446.
            Each set of params maps to the connector classname specified.
         </description>
         <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>
      <!--
      <config-property>
        <description>Does we support HA</description>
        <config-property-name>HA</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value>false</config-property-value>
      </config-property>
      <config-property>
        <description>The method to use for locating the
transactionmanager</description>
        <config-property-name>TransactionManagerLocatorMethod</config-
property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>getTm</config-property-value>
      </config-property>
      <config-property>
        <description>Use A local Transaction instead of XA?</description>
        <config-property-name>UseLocalTx</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value>false</config-property-value>
      </config-property>
      <config-property>
        <description>The user name used to login to the JMS
server</description>
        <config-property-name>UserName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The password used to login to the JMS
server</description>
        <config-property-name>Password</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The jndi params to use to look up the jms resources
if local jndi is not to be used</description>
        <config-property-name>JndiParams</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-
value>java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory;
java.naming.provider.url=jnp://localhost:1199;java.naming.factory.url.pkgs
=org.jboss.naming:org.jnp.interfaces</config-property-value>
      </config-property>
      <config-property>
        <description>The discovery group address</description>
        <config-property-name>DiscoveryAddress</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The discovery group port</description>
        <config-property-name>DiscoveryPort</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The discovery refresh timeout</description>
        <config-property-name>DiscoveryRefreshTimeout</config-property-
name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The discovery initial wait timeout</description>
        <config-property-name>DiscoveryInitialWaitTimeout</config-
property-name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The load balancing policy class name</description>
        <config-property-name>LoadBalancingPolicyClassName</config-
property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The client failure check period</description>
        <config-property-name>ClientFailureCheckPeriod</config-property-
name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The connection TTL</description>
        <config-property-name>ConnectionTTL</config-property-name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The call timeout</description>
        <config-property-name>CallTimeout</config-property-name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The dups ok batch size</description>
        <config-property-name>DupsOKBatchSize</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The transaction batch size</description>
        <config-property-name>TransactionBatchSize</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The consumer window size</description>
        <config-property-name>ConsumerWindowSize</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The consumer max rate</description>
        <config-property-name>ConsumerMaxRate</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The confirmation window size</description>
        <config-property-name>ConfirmationWindowSize</config-property-
name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The producer max rate</description>
        <config-property-name>ProducerMaxRate</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The min large message size</description>
        <config-property-name>MinLargeMessageSize</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The block on acknowledge</description>
        <config-property-name>BlockOnAcknowledge</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The block on non durable send</description>
        <config-property-name>BlockOnNonDurableSend</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The block on durable send</description>
        <config-property-name>BlockOnDurableSend</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The auto group</description>
        <config-property-name>AutoGroup</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The max connections</description>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The pre acknowledge</description>
        <config-property-name>PreAcknowledge</config-property-name>
        <config-property-type>java.lang.Boolean</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The retry interval</description>
        <config-property-name>RetryInterval</config-property-name>
        <config-property-type>java.lang.Long</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The retry interval multiplier</description>
        <config-property-name>RetryIntervalMultiplier</config-property-
name>
        <config-property-type>java.lang.Double</config-property-type>
        <config-property-value></config-property-value>
      </config-property>
      <config-property>
        <description>The client id</description>
        <config-property-name>ClientID</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value></config-property-value>
      </config-property>-->

      <outbound-resourceadapter>
         <connection-definition>
            <managedconnectionfactory-
class>org.hornetq.ra.HornetQRAManagedConnectionFactory</managedconnectionf
actory-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>
            <connectionfactory-impl-
class>org.hornetq.ra.HornetQRAConnectionFactoryImpl</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>
</connector>




  <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.InVMConnectorFactory
      </config-property-value>
   </config-property>
   <config-property>
      <description>
        The transport configuration. These values must be in the form of
key=val;key=val;
      </description>
      <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>
         <connectionfactory-impl-class>
   org.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>
```

APPENDIX C. REVISION HISTORY

Revision 5.2.0-100.400	**2013-10-30**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.2.0-100	**Wed 23 Jan 2013**	**Russell Dickenson**

Incorporated changes for JBoss Enterprise Application Platform 5.2.0 GA. For information about documentation changes to this guide, refer to *Release Notes 5.2.0*.

Revision 5.1.2-109	**Wed 18 Jul 2012**	**Anthony Towns**

Rebuild for Publican 3.

Revision 5.1.2-100	**Thu 8 Dec 2011**	**Russell Dickenson**

Incorporated changes for JBoss Enterprise Application Platform 5.1.2 GA. For information about documentation changes to this guide, refer to *Release Notes 5.1.2*.