AMQP Messaging Broker (Java)

AMQP Messaging Broker (Java)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

The Java Broker is a powerful open-source message broker that implements all versions of the Advanced Message Queuing Protocol (AMQP) [http://www.amqp.org]. The Java Broker is actually one of two message brokers provided by the Apache Qpid project [http://qpid.apache.org]: the Java Broker and the C++ Broker.

This document relates to the Java Broker. The C++ Broker is described separately [../../AMQP-Messaging-Broker-CPP-Book/html/].

Headline features

- 100% Java implementation - runs on any platform supporting Java 1.7 or higher
- Messaging clients support in Java, C++, Python.
- JMS 1.1 compliance (Java client).
- Persistent and non-persistent (transient) message support
- Supports for all common messaging patterns (point-to-point, publish/subscribe, fan-out etc).
- Transaction support including XA[1]
- Supports for all versions of the AMQP protocol
- Automatic message translation, allowing clients using different AMQP versions to communicate with each other.
- Pluggable authentication architecture with out-of-the-box support for Kerberos, LDAP, External, and file-based authentication mechanisms.
- Pluggable storage architecture with implementations including Apache Derby [http://db.apache.org/derby/], Oracle BDB JE [http://www.oracle.com/technetwork/products/berkeleydb/overview/index-093405.html][2], and External Database
- Web based management interface and programmatic management interfaces via REST and JMX APIs.
- SSL support
- High availability (HA) support.[3]

[1]XA provided when using AMQP 0-10
[2]Oracle BDB JE must be downloaded separately.
[3]HA currently only available to users of the optional BDB JE HA based message store.

Chapter 2. Installation

2.1. Introduction

This document describes how to install the Java Broker on both Windows and UNIX platforms.

2.2. Prerequisites

2.2.1. Java Platform

The Java Broker is an 100% Java implementation and as such it can be used on any operating system supporting Java 1.7 or higher[1]. This includes Linux, Solaris, Mac OS X, and Windows XP/Vista/7/8.

The broker has been tested with Java implementations from both Oracle and IBM. Whatever platform you chose, it is recommended that you ensure it is patched with any critical updates made available from the vendor.

Verify that your JVM is installed properly by following these instructions.

2.2.2. Disk

The Java Broker installation requires approximately 20MB of free disk space.

The Java Broker also requires a working directory. The working directory is used for the message store, that is, the area of the file-system used to record persistent messages whilst they are passing through the Broker. The working directory is also used for the default location of the log file. The size of the working directory will depend on the how the Broker is used.

The performance of the file system hosting the work directory is key to the performance of Broker as a whole. For best performance, choose a device that has low latency and one that is uncontended by other applications.

Be aware that there are additional considerations if you are considering hosting the working directory on NFS.

2.2.3. Memory

Qpid caches messages on the heap for performance reasons, so in general, the Broker will benefit from as much heap as possible. However, on a 32bit JVM, the maximum addressable memory range for a process is 4GB, after leaving space for the JVM's own use this will give a maximum heap size of approximately ~3.7GB.

2.2.4. Operating System Account

Installation or operation of Qpid does *not* require a privileged account (i.e. root on UNIX platforms or Administrator on Windows). However it is suggested that you use an dedicated account (e.g. qpid) for the installation and operation of the Java Broker.

[1]Java Cryptography Extension (JCE) Unlimited Strength required for some features

2.3. Download

2.3.1. Broker Release

You can download the latest Java broker package from the Download Page [http://qpid.apache.org/download.html].

It is recommended that you confirm the integrity of the download by verifying the PGP signature matches that available on the site. Instructions are given on the download page.

2.4. Installation on Windows

Firstly, verify that your JVM is installed properly by following these instructions.

Now chose a directory for Qpid broker installation. This directory will be used for the Qpid JARs and configuration files. It need not be the same location as the work directory used for the persistent message store or the log file (you will choose this location later). For the remainder this example we will assume that location c:\qpid has been chosen.

Next extract the qpid-java-broker-0.32-SNAPSHOT.zip package into the directory, using either the zip file handling offered by Windows (right click the file and select 'Extract All') or a third party tool of your choice.

The extraction of the broker package will have created a directory qpid-broker-0.32-SNAPSHOT within c:\qpid

```
 Directory of c:\qpid\qpid-broker-0.32-SNAPSHOT

07/25/2012  11:22 PM                  .
09/30/2012  10:51 AM                  ..
09/30/2012  12:24 AM                  bin
08/21/2012  11:17 PM                  etc
07/25/2012  11:22 PM                  lib
07/20/2012  08:10 PM          65,925 LICENSE
07/20/2012  08:10 PM           3,858 NOTICE
07/20/2012  08:10 PM           1,346 README.txt
```

2.4.1. Setting the working directory

Qpid requires a work directory. This directory is used for the default location of the Qpid log file and is used for the storage of persistent messages. The work directory can be set on the command-line (for the lifetime of the command interpreter), but you will normally want to set the environment variable permanently via the Advanced System Settings in the Control Panel.

```
set QPID_WORK=C:\qpidwork
```

If the directory referred to by QPID_WORK does not exist, the Java Broker will attempt to create it on start-up.

2.5. Installation on UNIX platforms

Firstly, verify that your JVM is installed properly by following these instructions.

Now chose a directory for Qpid broker installation. This directory will be used for the Qpid JARs and configuration files. It need not be the same location as the work directory used for the persistent message store or the log file (you will choose this location later). For the remainder this example we will assume that location /usr/local/qpid has been chosen.

Next extract the qpid-java-broker-0.32-SNAPSHOT.tar.gz package into the directory.

```
mkdir /usr/local/qpid
cd /usr/local/qpid
tar xvzf qpid-java-broker-0.32-SNAPSHOT.tar.gz
```

The extraction of the broker package will have created a directory qpid-broker-0.32-SNAPSHOT within /usr/local/qpid

```
ls -la qpid-broker-0.32-SNAPSHOT/
total 152
drwxr-xr-x   8 qpid  qpid    272 25 Jul 23:22 .
drwxr-xr-x  45 qpid  qpid   1530 30 Sep 10:51 ..
-rw-r--r--@  1 qpid  qpid  65925 20 Jul 20:10 LICENSE
-rw-r--r--@  1 qpid  qpid   3858 20 Jul 20:10 NOTICE
-rw-r--r--@  1 qpid  qpid   1346 20 Jul 20:10 README.txt
drwxr-xr-x  10 qpid  qpid    340 30 Sep 00:24 bin
drwxr-xr-x   9 qpid  qpid    306 21 Aug 23:17 etc
drwxr-xr-x  34 qpid  qpid   1156 25 Jul 23:22 lib
```

2.5.1. Setting the working directory

Qpid requires a work directory. This directory is used for the default location of the Qpid log file and is used for the storage of persistent messages. The work directory can be set on the command-line (for the lifetime of the current shell), but you will normally want to set the environment variable permanently the user's shell profile file (~/.bash_profile for Bash etc).

```
export QPID_WORK=/var/qpidwork
```

If the directory referred to by QPID_WORK does not exist, the Java Broker will attempt to create it on start-up.

2.6. Optional Dependencies

If you wish to utilise storage options using Oracle BDB JE or an External Database, see Section E.3, “Installing Oracle BDB JE” and Section E.2, “Installing External JDBC Driver” for details of installing their dependencies.

Chapter 3. Getting Started

3.1. Introduction

This section describes how to start and stop the Java Broker, and outlines the various command line options.

For additional details about the broker configuration store and related command line arguments see Chapter 5, *Initial Configuration*. The broker is fully configurable via its Web Management Console, for details of this see Section 6.2, “Web Management Console”.

3.2. Starting/Stopping the broker on Windows

Firstly change to the installation directory used during the installation and ensure that the QPID_WORK environment variable is set.

Now use the **qpid-server.bat** to start the server

```
bin\qpid-server.bat
```

Output similar to the following will be seen:

```
[Broker] BRK-1006 : Using configuration : C:\qpidwork\config.json
[Broker] BRK-1007 : Using logging configuration : C:\qpid\qpid-broker-0.32-SNAPSHO
[Broker] BRK-1001 : Startup : Version: 0.32-SNAPSHOT Build: 1478262
[Broker] BRK-1010 : Platform : JVM : Oracle Corporation version: 1.7.0_21-b11 OS :
[Broker] BRK-1011 : Maximum Memory : 1,060,372,480 bytes
[Broker] BRK-1002 : Starting : Listening on TCP port 5672
[Broker] MNG-1001 : JMX Management Startup
[Broker] MNG-1002 : Starting : RMI Registry : Listening on port 8999
[Broker] MNG-1002 : Starting : JMX RMIConnectorServer : Listening on port 9099
[Broker] MNG-1004 : JMX Management Ready
[Broker] MNG-1001 : Web Management Startup
[Broker] MNG-1002 : Starting : HTTP : Listening on port 8080
[Broker] MNG-1004 : Web Management Ready
[Broker] BRK-1004 : Qpid Broker Ready
```

The BRK-1004 message confirms that the Broker is ready for work. The MNG-1002 and BRK-1002 confirm the ports to which the Broker is listening (for HTTP/JMX management and AMQP respectively).

To stop the Broker, use Control-C or use the Shutdown MBean from the JMX management plugin.

3.3. Starting/Stopping the broker on Unix

Firstly change to the installation directory used during the installation and ensure that the QPID_WORK environment variable is set.

Now use the **qpid-server** script to start the server:

```
bin/qpid-server
```

Output similar to the following will be seen:

```
[Broker] BRK-1006 : Using configuration : /var/qpidwork/config.json
```

```
[Broker] BRK-1007 : Using logging configuration : /usr/local/qpid/qpid-broker-0.32
[Broker] BRK-1001 : Startup : Version: 0.32-SNAPSHOT Build: exported
[Broker] BRK-1010 : Platform : JVM : Sun Microsystems Inc. version: 1.6.0_32-b05 O
[Broker] BRK-1011 : Maximum Memory : 1,065,025,536 bytes
[Broker] BRK-1002 : Starting : Listening on TCP port 5672
[Broker] MNG-1001 : Web Management Startup
[Broker] MNG-1002 : Starting : HTTP : Listening on port 8080
[Broker] MNG-1004 : Web Management Ready
[Broker] MNG-1001 : JMX Management Startup
[Broker] MNG-1002 : Starting : RMI Registry : Listening on port 8999
[Broker] MNG-1002 : Starting : JMX RMIConnectorServer : Listening on port 9099
[Broker] MNG-1004 : JMX Management Ready
[Broker] BRK-1004 : Qpid Broker Ready
```

The BRK-1004 message confirms that the Broker is ready for work. The MNG-1002 and BRK-1002 confirm the ports to which the Broker is listening (for HTTP/JMX management and AMQP respectively).

To stop the Broker, use Control-C from the controlling shell, use the **bin/qpid.stop** script, use **kill -TERM <pid>**, or the Shutdown MBean from the JMX management plugin.

3.4. Log file

The Java Broker writes a log file to record both details of its normal operation and any exceptional conditions. By default the log file is written within the log subdirectory beneath the work directory - `$QPID_WORK/log/qpid.log` (UNIX) and `%QPID_WORK%\log\qpid.log` (Windows).

For details of how to control the logging, see Section 9.1, “Log Files”

3.5. Using the command line

The Java Broker understands a number of command line options which may be used to customise the configuration.

For additional details about the broker configuration and related command line arguments see Chapter 5, *Initial Configuration*. The broker is fully configurable via its Web Management Console, for details of this see Section 6.2, “Web Management Console”.

To see usage information for all command line options, use the `--help` option

```
bin/qpid-server --help

usage: Qpid [-cic <path>] [-h] [-icp <path>] [-l <file>] [-mm] [-mmhttp <port>]
            [-mmjmx <port>] [-mmpass <password>] [-mmqv] [-mmrmi <port>] [-os]
            [-sp <path>] [-st <type>] [-v] [-w <period>]
 -cic <path>                                   Create a copy of the initial confi
 --create-initial-config <path>                file, either to an optionally spec
                                               file path, or as initial-config.js
                                               in the current directory

 -h,                                           Print this message
 --help

 -icp  <path>                                  Set the location of initial JSON c
 --initial-config-path <path>                  to use when creating/overwriting a
```

```
                                                 broker configuration store

-l <file>                                        Use the specified log4j xml config
--logconfig <file>                               file. By default looks for a file
                                                 etc/log4j.xml in the same director
                                                 the configuration file

-mm                                              Start broker in management mode,
--management-mode                                disabling the AMQP ports

-mmhttp <port>                                   Override http management port in
--management-mode-http-port <port>               management mode

-mmjmx                                           Override jmx connector port in
--management-mode-jmx-connector-port <port>      management mode

-mmpass  <password>                              Set the password for the managemen
--management-mode-password <password>            mode user mm_admin

-mmqv                                            Make virtualhosts stay in the quie
--management-mode-quiesce-virtualhosts           state during management mode.

-mmrmi <port>                                    Override jmx rmi registry port in
--management-mode-rmi-registry-port <port>       management mode

-os                                              Overwrite the broker configuration
--overwrite-store                                with the current initial configura

-prop "<name=value>"                             Set a configuration property to us
--config-property "<name=value>"                 resolving variables in the broker
                                                 configuration store, with format
                                                 "name=value"

-sp <path>                                       Use given configuration store loca
--store-path <path>

-st <type>                                       Use given broker configuration sto
--store-type <type>

-v                                               Print the version information and
--version

-w <period>                                      Monitor the log file configuration
--logwatch <period>                              for changes. Units are seconds. Ze
                                                 means do not check for changes.
```

Chapter 4. Concepts

4.1. Overview

The Broker comprises of a number of entities. This section summaries the purpose of each of the entities and describes the relationships between them. These details are developed further in the sub-sections that follow.

The most important entity is the *Virtualhost*. A virtualhost is an independent namespace in which messaging is performed. A *virtualhost* exists in a container called a *virtualhost node*. A virtualhost node has exactly one virtualhost.

Ports accept connections for messaging and management. The Broker supports any number of ports. When connecting for messaging, the user specifies a virtualhost name to indicate the virtualhost to which it is to be connected.

Authentication Providers assert the identity of the user as it connects for messaging or management. The Broker supports any number of authentication providers. Each port is associated with exactly one authentication provider. The port uses the authentication provider to assert the identity of the user as new connections are received.

Group Providers provide mechanisms that provide grouping of users. A Broker supports zero or more group providers.

Access Control Provider allows the abilities of users (or groups of users) to be restrained. A Broker can have zero or one access control providers.

Keystores provide a repositories of certificates and are used when the Broker accepts SSL connections. Any number of keystore providers can be defined. Keystores are be associated with Ports defined to accepts SSL.

Truststores provide a repositories of trust and are used to validate a peer. Any number of truststore provides can be defined. Truststores can be associated with Ports and other entities that form SSL connections.

Remote Replication Nodes are used when the high availability feature is in use. It is the remote representation of other virtualhost nodes that form part of the same group.

These concepts will be developed over the forthcoming pages. The diagrams below also help put these entities in context of one and other.

Figure 4.1. Message Flow through Key Entities

Figure 4.2. Broker Structure

4.2. Broker

The *Broker* is the outermost entity within the system.

The Broker is backed by storage. This storage is used to record the durable entities that exist beneath it.

4.3. Virtualhost Nodes

A *virtualhost node* is a container for the virtualhost. It has exactly one virtualhost.

A *virtualhost node* is backed by storage. This storage is used to record the durable entities that exist beneath the virtualhost node (the virtualhost, queues, exchanges etc).

When HA is in use, it is the virtualhost nodes of many Brokers that come together to form the group. The virtualhost nodes together elect a master. When the high availability feature is in use, the virtualhost node has remote replications nodes. There is a remote replication node corresponding to each remote virtualhost node that form part of the group.

4.4. Remote Replication Nodes

Used for HA only. A *remote replication node* is a representation of another virtualhost node in the group.

4.5. Virtualhosts

A virtualhost is a namespace in which messaging is performed. Virtualhosts are independent; the messaging that goes on within one virtualhost is independent of any messaging that goes on in another virtualhost. For instance, a queue named *foo* defined in one virtualhost is completely independent of a queue named *foo* in another virtualhost.

A virtualhost is identified by a name which must be unique broker-wide. Clients use the name to identify the virtualhost to which they wish to connect when they connect.

A virtualhost exists in a container called a virtualhost node.

The virtualhost comprises of a number of entities. This section summaries the purpose of each of the entities and describes the relationships between them. These details are developed further in the sub-sections that follow.

Exchanges is a named entity within the Virtual Host which receives messages from producers and routes them to matching Queues.

Queues are named entities that hold messages for delivery to consumer applications.

Bindings are relationships between Exchanges and Queue that facilitate routing of messages from the Exchange to the Queue.

Connections represent a live connection to the virtualhost from a messaging client.

A *Session* represents a context for the production or consumption of messages. A Connection can have many Sessions.

A *Consumer* represents a live consumer that is attached to queue.

The following diagram depicts the Virtualhost model:

Figure 4.3. Virtualhost Model

A *virtualhost* is backed by storage which is used to store the messages.

4.6. Exchanges

An *Exchange* is a named entity within the *Virtualhost* which receives messages from producers and routes them to matching *Queue*s within the *Virtualhost*.

The server provides a set of exchange types with each exchange type implementing a different routing algorithm. For details of how these exchanges types work see Section 4.6.2, “Exchange Types” below.

The server predeclares a number of exchange instances with names starting with "`amq.`". These are defined in Section 4.6.1, “Predeclared Exchanges”.

Applications can make use of the pre-declared exchanges, or they may declare their own. The number of exchanges within a *Virtualhost* is limited only by resource constraints.

The behaviour when an *Exchange* is unable to route a message to any queue is defined in Section 4.6.4, “Unrouteable Messages”

4.6.1. Predeclared Exchanges

Each *Virtualhost* pre-declares the following exchanges:

- amq.direct (an instance of a direct exchange)
- amq.topic (an instance of a topic exchange)
- amq.fanout (an instance of a fanout exchange)
- amq.match (an instance of a headers exchange)

The conceptual "`default exchange`" always exists, effectively a special instance of direct exchange which uses the empty string as its name. All queues are automatically bound to it upon their creation using the queue name as the binding key, and unbound upon their deletion. It is not possible to manually add or remove bindings within this exchange.

Applications may not declare exchanges with names beginning with "`amq.`". Such names are reserved for system use.

4.6.2. Exchange Types

The following Exchange types are supported.

- Direct
- Topic
- Fanout
- Headers

These exchange types are described in the following sub-sections.

4.6.2.1. Direct

The direct exchange type routes messages to queues based on an exact match between the routing key of the message, and the binding key used to bind the queue to the exchange. Additional filter rules may be specified using a binding argument specifying a JMS message selector.

This exchange type is often used to implement point to point messaging. When used in this manner, the normal convention is that the binding key matches the name of the queue. It is also possible to use this exchange type for multi-cast, in this case the same binding key is associated with many queues.

Figure 4.4. Direct exchange

The figure above illustrates the operation of direct exchange type. The yellow messages published with the routing key "`myqueue`" match the binding key corresponding to queue "`myqueue`" and so are routed there. The red messages published with the routing key "`foo`" match two bindings in the table so a copy of the message is routed to both the "`bar1`" and "`bar2`" queues.

The routing key of the blue message matches no binding keys, so the message is unroutable. It is handled as described in Section 4.6.4, “Unrouteable Messages”.

4.6.2.2. Topic

This exchange type is used to support the classic publish/subscribe paradigm.

The topic exchange is capable of routing messages to queues based on wildcard matches between the routing key and the binding key pattern defined by the queue binding. Routing keys are formed from one or more words, with each word delimited by a full-stop (.). The pattern matching characters are the * and # symbols. The * symbol matches a single word and the # symbol matches zero or more words.

Additional filter rules may be specified using a binding argument specifying a JMS message selector.

The following three figures help explain how the topic exchange functions.

Figure 4.5. Topic exchange - exact match on topic name

The figure above illustrates publishing messages with routing key "`weather`". The exchange routes each message to every bound queue whose binding key matches the routing key.

In the case illustrated, this means that each subscriber's queue receives every yellow message.

Figure 4.6. Topic exchange - matching on hierarchical topic patterns

The figure above illustrates publishing messages with hierarchical routing keys. As before, the exchange routes each message to every bound queue whose binding key matches the routing key but as the binding keys contain wildcards, the wildcard rules described above apply.

In the case illustrated, `sub1` has received the red and green message as "`news.uk`" and "`news.de`" match binding key "`news.#`". The red message has also gone to `sub2` and `sub3` as it's routing key is matched exactly by "`news.uk`" and by "`*.uk`".

The routing key of the yellow message matches no binding keys, so the message is unroutable. It is handled as described in Section 4.6.4, “Unrouteable Messages”.

Figure 4.7. Topic exchange - matching on JMS message selector

The figure above illustrates messages with properties published with routing key "`shipping`".

As before, the exchange routes each message to every bound queue whose binding key matches the routing key but as a JMS selector argument has been specified, the expression is evaluated against each matching message. Only messages whose message header values or properties match the expression are routed to the queue.

In the case illustrated, `sub1` has received the yellow and blue message as their property "`area`" cause expression "`area in ('Forties', 'Cromarty')`" to evaluate true. Similarly, the yellow message has also gone to `gale_alert` as its property "`speed`" causes expression "`speed > 7 and speed < 10`" to evaluate true.

The properties of purple message cause no expressions to evaluate true, so the message is unroutable. It is handled as described in Section 4.6.4, “Unrouteable Messages”.

4.6.2.3. Fanout

The fanout exchange type routes messages to all queues bound to the exchange, regardless of the message's routing key.

Filter rules may be specified using a binding argument specifying a JMS message selector.

Figure 4.8. Fanout exchange

4.6.2.4. Headers

The headers exchange type routes messages to queues based on header properties within the message. The message is passed to a queue if the header properties of the message satisfy the x-match expression specified by the binding arguments with which the queue was bound.

4.6.3. Binding Arguments

Binding arguments are used by certain exchange types to further filter messages.

4.6.3.1. JMS Selector

The binding argument `x-filter-jms-selector` specifies a JMS selector conditional expression. The expression is written in terms of message header and message property names. If the expression evaluates to true, the message is routed to the queue. This type of binding argument is understood by exchange types direct, topic and fanout.[1].

4.6.3.2. x-match

The binding argument `x-match` is understood by exchange type headers. It can take two values, dictating how the rest of the name value pairs are treated during matching.

- `all` implies that all the other pairs must match the headers property of a message for that message to be routed (i.e. an AND match)
- `any` implies that the message should be routed if any of the fields in the headers property match one of the fields in the arguments table (i.e. an OR match)

[1] This is a Qpid specific extension.

A field in the bind arguments matches a field in the message if either the field in the bind arguments has no value and a field of the same name is present in the message headers or if the field in the bind arguments has a value and a field of the same name exists in the message headers and has that same value.

4.6.4. Unrouteable Messages

If an exchange is unable to route a message to any queues, the Broker will:

- If using AMQP 0-10 protocol, and an alternate exchange has been set on the exchange, the message is routed to the alternate exchange. The alternate exchange routes the message according to its routing algorithm and its binding table. If the messages is still unroutable, the message is discarded.
- If using AMQP protocols 0-8..0-9-1, and the publisher set the mandatory flag and the close when no route feature did not close the connection, the message is returned to the Producer.
- Otherwise, the message is discarded.

4.7. Queues

*Queue*s are named entities within a Virtualhost that hold/buffer messages for later delivery to consumer applications. An Exchange for passing messages to a queue. Consumers subscribe to a queue in order to receive messages for it.

The Broker supports different queue types, each with different delivery semantics. It also messages on a queue to be treated as a group.

4.7.1. Types

The Broker supports four different queue types, each with different delivery semantics.

- Standard - a simple First-In-First-Out (FIFO) queue
- Priority - delivery order depends on the priority of each message
- Sorted - delivery order depends on the value of the sorting key property in each message
- Last Value Queue - also known as an LVQ, retains only the last (newest) message received with a given LVQ key value

4.7.1.1. Standard

A simple First-In-First-Out (FIFO) queue

4.7.1.2. Priority

In a priority queue, messages on the queue are delivered in an order determined by the JMS priority message header [http://docs.oracle.com/javaee/6/api/javax/jms/Message.html#getJMSPriority()] within the message. By default Qpid supports the 10 priority levels mandated by JMS, with priority value 0 as the lowest priority and 9 as the highest.

It is possible to reduce the effective number of priorities if desired.

JMS defines the default message priority [http://docs.oracle.com/javaee/6/api/javax/jms/Message.html#DEFAULT_PRIORITY] as 4. Messages sent without a specified priority use this default.

4.7.1.3. Sorted Queues

Sorted queues allow the message delivery order to be determined by value of an arbitrary JMS message property [http://docs.oracle.com/javaee/6/api/javax/jms/Message.html#getStringProperty()]. Sort order is alpha-numeric and the property value must have a type java.lang.String.

Messages sent to a sorted queue without the specified JMS message property will be inserted into the 'last' position in the queue.

4.7.1.4. Last Value Queues (LVQ)

LVQs (or conflation queues) are special queues that automatically discard any message when a newer message arrives with the same key value. The key is specified by arbitrary JMS message property [http://docs.oracle.com/javaee/6/api/javax/jms/Message.html#getPropertyNames()].

An example of an LVQ might be where a queue represents prices on a stock exchange: when you first consume from the queue you get the latest quote for each stock, and then as new prices come in you are sent only these updates.

Like other queues, LVQs can either be browsed or consumed from. When browsing an individual subscriber does not remove the message from the queue when receiving it. This allows for many subscriptions to browse the same LVQ (i.e. you do not need to create and bind a separate LVQ for each subscriber who wishes to receive the contents of the LVQ).

Messages sent to an LVQ without the specified property will be delivered as normal and will never be "replaced".

4.7.2. Queue Declare Arguments

To create a priority, sorted or LVQ queue programmatically from JMX or AMQP, pass the appropriate queue-declare arguments.

Table 4.1. Queue-declare arguments understood for priority, sorted and LVQ queues

Queue type	Argument name	Argument name	Argument Description
priority	x-qpid-priorities	java.lang.Integer	Specifies a priority queue with given number priorities
sorted	qpid.queue_sort_key	java.lang.String	Specifies sorted queue with given message property used to sort the entries
lvq	qpid.last_value_queue_key	java.lang.String	Specifies lvq queue with given message property used to conflate the entries

4.7.3. Messaging Grouping

The broker allows messaging applications to classify a set of related messages as belonging to a group. This allows a message producer to indicate to the consumer that a group of messages should be considered a single logical operation with respect to the application.

The broker can use this group identification to enforce policies controlling how messages from a given group can be distributed to consumers. For instance, the broker can be configured to guarantee all the messages from a particular group are processed in order across multiple consumers.

For example, assume we have a shopping application that manages items in a virtual shopping cart. A user may add an item to their shopping cart, then change their mind and remove it. If the application sends an *add* message to the broker, immediately followed by a *remove* message, they will be queued in the proper order - *add*, followed by *remove*.

However, if there are multiple consumers, it is possible that once a consumer acquires the *add* message, a different consumer may acquire the *remove* message. This allows both messages to be processed in parallel, which could result in a "race" where the *remove* operation is incorrectly performed before the *add* operation.

4.7.3.1. Grouping Messages

In order to group messages, the application would designate a particular message header as containing a message's *group identifier*. The group identifier stored in that header field would be a string value set by the message producer. Messages from the same group would have the same group identifier value. The key that identifies the header must also be known to the message consumers. This allows the consumers to determine a message's assigned group.

The header that is used to hold the group identifier, as well as the values used as group identifiers, are totally under control of the application.

4.7.3.2. The Role of the Broker in Message Grouping

The broker will apply the following processing on each grouped message:

- Enqueue a received message on the destination queue.
- Determine the message's group by examining the message's group identifier header.
- Enforce *consumption ordering* among messages belonging to the same group. *Consumption ordering* means one of two things depending on how the queue has been configured.
  - In default mode, a group gets assigned to a single consumer for the lifetime of that consumer, and the broker will pass all subsequent messages in the group to that consumer.
  - In 'shared groups' mode (which gives the same behaviour as the Qpid C++ Broker) the broker enforces a looser guarantee, namely that all the *currently unacknowledged messages* in a group are sent to the same consumer, but the consumer used may change over time even if the consumers do not. This means that only one consumer can be processing messages from a particular group at any given time, however if the consumer acknowledges all of its acquired messages then the broker *may* pass the next pending message in that group to a different consumer.

The absence of a value in the designated group header field of a message is treated as follows:

- In default mode, failure for a message to specify a group is treated as a desire for the message not to be grouped at all. Such messages will be distributed to any available consumer, without the ordering quarantees imposed by grouping.
- In 'shared groups' mode (which gives the same behaviour as the Qpid C++ Broker) the broker assigns messages without a group value to a 'default group'. Therefore, all such "unidentified" messages are considered by the broker as part of the same group, which will handled like any other group. The name of this default group is "qpid.no-group", although it can be customised as detailed below.

Note that message grouping has no effect on queue browsers.

Note well that distinct message groups would not block each other from delivery. For example, assume a queue contains messages from two different message groups - say group "A" and group "B" - and they are enqueued such that "A"'s messages are in front of "B". If the first message of group "A" is in the process of being consumed by a client, then the remaining "A" messages are blocked, but the messages of the "B" group are available for consumption by other consumers - even though it is "behind" group "A" in the queue.

4.7.4. Using low pre-fetch with special queue types

Qpid clients receive buffered messages in batches, sized according to the pre-fetch value. The current default is 500.

However, if you use the default value you will probably *not* see desirable behaviour when using priority, sorted, lvq or grouped queues. Once the broker has sent a message to the client its delivery order is then fixed, regardless of the special behaviour of the queue.

For example, if using a priority queue and a prefetch of 100, and 100 messages arrive with priority 2, the broker will send these messages to the client. If then a new message arrives with priority 1, the broker cannot leap frog messages of lower priority. The priority 1 will be delivered at the front of the next batch of messages to be sent to the client.

So, you need to set the prefetch values for your client (consumer) to make this sensible. To do this set the Java system property `max_prefetch` on the client environment (using -D) before creating your consumer.

A default for all client connections can be set via a system property:

```
-Dmax_prefetch=1
```

The prefetch can be also be adjusted on a per connection basis by adding a `maxprefetch` value to the Connection URLs [../../Programming-In-Apache-Qpid/html/QpidJNDI.html#section-jms-connection-url]

```
amqp://guest:guest@client1/development?maxprefetch='1'&brokerlist='tcp://localhost
```

Setting the Qpid pre-fetch to 1 will give exact queue-type semantics as perceived by the client however, this brings a performance cost. You could test with a slightly higher pre-fetch to trade-off between throughput and exact semantics.

4.7.5. Forcing all consumers to be non-destructive

When a consumer attaches to a queue, the normal behaviour is that messages are sent to that consumer are acquired exclusively by that consumer, and when the consumer acknowledges them, the messages are removed from the queue.

Another common pattern is to have queue "browsers" which send all messages to the browser, but do not prevent other consumers from receiving the messages, and do not remove them from the queue when the browser is done with them. Such a browser is an instance of a "non-destructive" consumer.

If every consumer on a queue is non destructive then we can obtain some interesting behaviours. In the case of a LVQ then the queue will always contain the most up to date value for every key. For a standard queue, if every consumer is non-destructive then we have something that behaves like a topic (every consumer receives every message) except that instead of only seeing messages that arrive after the point at which the consumer is created, all messages which have not been removed due to TTL expiry (or, in the case of LVQs, overwirtten by newer values for the same key).

A queue can be created to enforce all consumers are non-destructive. This can be be achieved using the following queue declare argument:

Table 4.2.

Argument Name	**Argument Type**	**Argument Description**
qpid.ensure_nondestructive_consumers	java.lang.Boolean	Set to true if the queue should make all consumers attached to it behave non-destructively. (Default is false).

Through the REST api, the equivalent attribute is named `ensureNondestructiveConsumers`.

4.7.5.1. Bounding size using min/max TTL

For queues other than LVQs, having only non-destructive consumers could mean that messages would never get deleted, leaving the queue to grow unconstrainedly. To prevent this you can use the ability to set the maximum TTL of the queue. To ensure all messages have the same TTL you could also set the minimum TTL to the same value.

Minimum/Maximum TTL for a queue can be set though the HTTP Management UI, using the REST API or by hand editing the configuration file (for JSON configuration stores). The attribute names are `minimumMessageTtl` and `maximumMessageTtl` and the TTL value is given in milliseconds.

4.7.5.2. Choosing to receive messages based on arrival time

A queue with no destructive consumers will retain all messages until they expire due to TTL. It may be the case that a consumer only wishes to receive messages that have been sent in the last 60 minutes, and any new messages that arrive, or alternatively it may wish only to receive newly arriving messages and not any that are already in the queue. This can be achieved by using a filter on the arrival time.

A special parameter `x-qpid-replay-period` can be used in the consumer declaration to control the messages the consumer wishes to receive. The value of `x-qpid-replay-period` is the time, in seconds, for which the consumer wishes to see messages. A replay period of 0 indicates only newly arriving messages should be sent. A replay period of 3600 indicates that only messages sent in the last hour - along with any newly arriving messages - should be sent.

Table 4.3. Setting the replay period

Syntax	**Example**
Addressing	myqueue : { link : { x-subscribe: { arguments : { x-qpid-replay-period : '3600' } } } }
Binding URL	direct://amq.direct/myqueue/myqueue?x-qpid-replay-period='3600'

4.7.5.3. Setting a default filter

A common case might be that the desired default behaviour is that newly attached consumers see only newly arriving messages (i.e. standard topic-like behaviour) but other consumers may wish to start their message stream from some point in the past. This can be achieved by setting a default filter on the queue so that consumers which do not explicitly set a replay period get a default (in this case the desired default would be 0).

The default filter set for a queue can be set via the REST API using the attribute named `defaultFilters`. This value is a map from filter name to type and arguments. To set the default behaviour for the queue to be that consumers only receive newly arrived messages, then you should set this attribute to the value:

```
{ "x-qpid-replay-period" : { "x-qpid-replay-period" : [ "0" ] } }
```

If the desired default behaviour is that each consumer should see all messages arriving in the last minute, as well as all new messages then the value would need to be:

```
{ "x-qpid-replay-period" : { "x-qpid-replay-period" : [ "60" ] } }
```

4.8. Ports

The Broker supports configuration of *Ports* to specify the particular AMQP messaging and HTTP/JMX management connectivity it offers for use.

Each Port is configured with the particular *Protocols* and *Transports* it supports, as well as the *Authentication Provider* to be used to authenticate connections. Where SSL is in use, the *Port* configuration also defines which *Keystore* to use and (where supported) which *TrustStore(s)* and whether Client Certificates should be requested/required.

Different *Ports* can support different protocols, and many *Ports* can be configured on the Broker.

The following AMQP protocols are currently supported by the Broker:

- *AMQP 0-8*
- *AMQP 0-9*
- *AMQP 0-9-1*
- *AMQP 0-10*
- *AMQP 1.0*

Additionally, HTTP and JMX ports can be configured for use by the associated management plugins.

This diagram explains how Ports, Authentication Providers and an Access Control Provider work together to allow an application to form a connection to a Virtualhost.

Figure 4.9. Control flow during Authentication

4.9. Authentication Providers

Authentication Providers are used by *Ports* to authenticate connections. Many *Authentication Providers* can be configured on the Broker at the same time, from which each *Port* can be assigned one.

Some Authentication Providers offer facilities for creation and deletion of users.

4.10. Other Services

The Broker can also have *Access Control Providers*, *Group Providers*, *Keystores*, *Trustores* and [Management] *Plugins* configured.

4.10.1. Access Control Providers

Access Control Providers are used to authorize various operations relating to Broker objects.

Access Control Provider configuration and management details are covered in Section 8.3, “Access Control Lists”.

4.10.2. Group Providers

Group Providers are used to aggregate authenticated user principals into groups which can be then be used in Access Control rules applicable to the whole group.

Group Provider configuration and management is covered in Section 8.2, “Group Providers”.

4.10.3. Keystores

Keystores are used to configure details of keystores holding SSL keys and certificates for the SSL transports on Ports.

Keystore configuration and management is covered in Section 7.12, “Keystores”.

4.10.4. Truststores

Truststores are used to configure details of keystores holding SSL certificates for trusting Client Certificate on SSL ports.

Truststore configuration and management is covered in Section 7.13, “Truststores”.

Chapter 5. Initial Configuration

5.1. Introduction

This section describes how to perform initial configuration on the command line. Once the Broker is started, subsequent management is performed using the Management interfaces

The configuration for each component is stored as an entry in the broker configuration store, currently implemented as a JSON file which persists changes to disk, BDB or Derby database or an in-memory store which does not. The following components configuration is stored there:

- Broker
- Virtual Host
- Port
- Authentication Provider
- Access Control Provider
- Group Provider
- Key store
- Trust store
- Plugin

Broker startup involves two configuration related items, the 'Initial Configuration' and the Configuration Store. When the broker is started, if a Configuration Store does not exist at the current store location then one will be initialised with the current 'Initial Configuration'. Unless otherwise requested to overwrite the configuration store then subsequent broker restarts will use the existing configuration store and ignore the contents of the 'Initial Configuration'.

5.2. Configuration Store Location

The broker will default to using ${qpid.work_dir}/config.json as the path for its configuration store unless otherwise instructed.

The command line argument *-sp* (or *--store-path*) can optionally be used to specify a different relative or absolute path to use for the broker configuration store:

```
$ ./qpid-server -sp ./my-broker-configuration.json
```

If no configuration store exists at the specified/defaulted location when the broker starts then one will be initialised using the current 'Initial Configuration'.

5.3. 'Initial Configuration' Location

The 'Initial Configuration' JSON file is used when initialising new broker configuration stores. The broker will default to using an internal file within its jar unless otherwise instructed.

The command line argument *-icp* (or *--initial-config-path*) can be used to override the brokers internal file and supply a user-created one:

```
$ ./qpid-server -icp ./my-initial-configuration.json
```

If a Configuration Store already exists at the current store location then the current 'Initial Configuration' will be ignored unless otherwise requested to overwrite the configuration store

5.4. Creating an 'Initial Configuration' JSON File

It is possible to have the broker output its default internal 'Initial Configuration' file to disk using the command line argument *-cic* (or *--create-initial-config*). If the option is used without providing a path, a file called *initial-config.json* will be created in the current directory, or alternatively the file can be created at a specified location:

```
$ ./qpid-server -cic ./initial-config.json
```

The 'Initial Configuration' JSON file shares a common format with the brokers JSON Configuration Store implementation, so it is possible to use a brokers Configuration Store output as an initial configuration. Typically 'Initial Configuration' files would not to contain IDs for the configured entities, so that IDs will be generated when the configuration store is initialised and prevent use of the same IDs across multiple brokers, however it may prove useful to include IDs if using the Memory Configuration Store Type.

It can be useful to use Configuration Properties within 'Initial Configuration' files to allow a degree of customisation with an otherwise fixed file.

For an example file, see Section 5.8, “Example of JSON 'Initial Configuration'”

5.5. Overwriting An Existing Configuration Store

If a configuration store already exists at the configured store location then it is used and the current 'Initial Configuration' is ignored.

The command line argument *-os* (or *--overwrite-store*) can be used to force a new broker configuration store to be initialised from the current 'Initial Configuration' even if one exists:

```
$ ./qpid-server -os -icp ./my-initial-configuration.json
```

This can be useful to effectively play configuration into one or more broker to pre-configure them to a particular state, or alternatively to ensure a broker is always started with a fixed configuration. In the latter case, use of the Memory Configuration Store Type may also be useful.

5.6. Configuration Store Type

There are currently two implementations of the pluggable Broker Configuration Store, the default one which persists content to disk in a JSON file, and another which operates only in-memory and so does not retain changes across broker restarts and always relies on the current 'Initial Configuration' to provide the configuration to start the broker with.

The command line argument *-st* (or *--store-type*) can be used to override the default *json*)configuration store type and allow choosing an alternative, such as *memory*)

```
$ ./qpid-server -st memory
```

This can be useful when running tests, or always wishing to start the broker with the same 'Initial Configuration'

5.7. Customising Configuration using Configuration Properties

It is possible for 'Initial Configuration' (and Configuration Store) files to contain ${properties} that can be resolved to String values at startup, allowing a degree of customisation using a fixed file. Configuration Property values can be set either via Java System Properties, or by specifying ConfigurationPproperties on the broker command line. If both are defined, System Property values take precedence.

The broker has the following set of core configuration properties, with the indicated default values if not otherwise configured by the user:

Table 5.1. Base Configuration Properties

Name	Description	Value
qpid.amqp_port	Port number used for the brokers default AMQP messaging port	"5672"
qpid.http_port	Port number used for the brokers default HTTP management port	"8080"
qpid.rmi_port	Port number used for the brokers default RMI Registry port, to advertise the JMX ConnectorServer.	"8999"
qpid.jmx_port	Port number used for the brokers default JMX port	"9099"
qpid.home_dir	Location of the broker installation directory, which contains the 'lib' directory and the 'etc' directory often used to store files such as group and ACL files.	Defaults to the value set into the QPID_HOME system property if it is set, or remains unset otherwise unless configured by the user.
qpid.work_dir	Location of the broker working directory, which might contain the persistent message store and broker configuration store files.	Defaults to the value set into the QPID_WORK system property if it is set, or the 'work' subdirectory

Name	Description	Value
		of the JVMs current working directory.

Use of these core properties can be seen in the default 'Initial Configuration' example.

Configuration Properties can be set on the command line using the *-prop* (or *--configuration-property*) command line argument:

```
$ ./qpid-server -prop "qpid.amqp_port=10000" -prop "qpid.http_port=10001"
```

In the example above, property used to set the port number of the default AMQP port is specified with the value 10000, overriding the default value of 5672, and similarly the value 10001 is used to override the default HTTP port number of 8080. When using the 'Initial Configuration' to initialise a new Configuration Store (either at first broker startup, when requesting to overwrite the configuration store) these new values will be used for the port numbers instead.

NOTE: When running the broker on Windows and starting it via the qpid-server.bat file, the "name=value" argument MUST be quoted.

5.8. Example of JSON 'Initial Configuration'

An example of the default 'Initial Configuration' JSON file the broker uses is provided below:

Example 5.1. JSON 'Initial configuration' File

```
{
  "name": "${broker.name}",
  "modelVersion": "2.0",
  "defaultVirtualHost" : "default",
  "authenticationproviders" : [ {
    "name" : "passwordFile",
    "type" : "PlainPasswordFile",
    "path" : "${qpid.home_dir}${file.separator}etc${file.separator}passwd",
    "preferencesproviders" : [{
        "name": "fileSystemPreferences",
        "type": "FileSystemPreferences",
        "path" : "${qpid.work_dir}${file.separator}user.preferences.json"
    }]
  } ],
  "ports" : [  {
    "name" : "AMQP",
    "port" : "${qpid.amqp_port}",
    "authenticationProvider" : "passwordFile"
  }, {
    "name" : "HTTP",
    "port" : "${qpid.http_port}",
    "authenticationProvider" : "passwordFile",
    "protocols" : [ "HTTP" ]
  }, {
    "name" : "RMI_REGISTRY",
    "port" : "${qpid.rmi_port}",
    "protocols" : [ "RMI" ]
  }, {
    "name" : "JMX_CONNECTOR",
    "port" : "${qpid.jmx_port}",
    "authenticationProvider" : "passwordFile",
    "protocols" : [ "JMX_RMI" ]
  }],
  "virtualhostnodes" : [ {
    "name" : "default",
    "type" : "JSON",
    "virtualHostInitialConfiguration" : "{ \"type\" : \"DERBY\" }"
  } ],
  "plugins" : [ {
    "type" : "MANAGEMENT-HTTP",
    "name" : "httpManagement"
  }, {
    "type" : "MANAGEMENT-JMX",
    "name" : "jmxManagement"
  } ]
}
```

In the configuration above the following entries are stored:

- Authentication Provider of type *PlainPasswordFile* with name "passwordFile".
- Four Port entries: "AMQP", "HTTP", "RMI_REGISTRY", "JMX_CONNECTOR".
- Virtualhost Node called default. On initial startup, it virtualHostInitialConfiguration will cause a virtualhost to be created with the same name. The configuration will be stored in a *JSON* configuration store, the message data will be stored in a *DERBY* message store.
- Two management plugins: "jmxManagement" of type "MANAGEMENT-JMX" and "httpManagement" of type "MANAGEMENT-HTTP".
- Broker attributes are stored as a root entry.

Chapter 6. Management Channels

The Broker can be managed over a number of different channels.

- HTTP - The primary channel for management. The HTTP interface comprises of a Web Console and a REST API.
- JMX - The Broker provides a JMX compliant management interface. This is not currently undergoing further development and is retained largely for backward compatibility. It is suggested the new users favour the Web Console/REST API.
- AMQP - The AMQP protocols 0-8..0-10 allow for some management of Exchanges, Queue and Bindings. This will be superseded by AMQP 1.0 Management. It is suggested that new users favour the Management facilities provided by the Web Console/REST API.

6.1. HTTP Management

6.1.1. Introduction

The HTTP Management plugin provides a HTTP based API for monitoring and control of the Broker. The plugin actually provides two interfaces:

- Web Management Console - rich web based interface for the management of the Broker.
- REST API - REST API providing complete programmatic management of the Broker.

The Web Management Console itself uses the REST API, so every function you can perform through the Web Management Console can be also be scripted and integrated into other systems. This provides a simple integration point allowing the Broker to monitored and controlled from systems such as Naoios or BMC Control-M.

6.1.2. Default Configuration

By default, the Broker is shipped with HTTP enabled running port 8080. The HTTP plugin is configured to require SASL authentication. The port is not SSL protected.

The settings can be changed by configuring the HTTP plugin and/or the port configured to serve HTTP.

6.2. Web Management Console

The Web Management Console provides a simple and intuitive interface for the Management and Control of the Broker. From here, all aspects of the Broker can be controlled, including:

- add, remove and monitor queues
- inspect, move, copy or delete messages
- add, remove and monitor virtualhosts
- configure and control high availability

The remainder of the section provides an introduction to the web management console and its use.

6.2.1. Accessing the Console

The Web Management Console is provided by the HTTP Management Plugin. Providing the HTTP Management Plugin is in its default configuration, the Web Management Console can be accessed by pointing a browser at the following URL:

```
http://myhost.mydomain.com:8080
```

The Console will prompt you to login using a username and password.

Figure 6.1. Web Management Console - Authentication

6.2.2. Orientation

After you have logged on you will see a screen similar to the following. The elements of the screen are now explained.

Figure 6.2. Web Management Orientation - Console

- *A* - Hierarchy view. Expandable/collapsible view showing all entities within the Broker. Double click on an entity name to cause its tab to be opened.
- *B* - Tab. Shows the details of an entity including its attributes and its child entities.
- *C* - Occluded tab. Click tab name to bring the tab to the front.
- *D* - Auto restore check box. Checked tabs will be automatically restored on subsequent login.
- *E* - Close. Click to close the tab.
- *F* - User Menu. Access to Preferences, Logout and Help.

Figure 6.3. Web Management Orientation - Tab

The elements of a tab are now explained:

- *1* - Attribute Panel. Shows the attributes of the entity. Click the panel title bar opens/closes the panel.
- *2* - Child Panels. Panels containing a table listing the children of the entity. Click the panel title bar opens/closes the panel.
- *3* - Child Row. Row summarizing a child entity. Double click to open the child tab.
- *4* - Child Operations. Buttons to add a new child or perform operations on existing children.

6.2.3. Managing Entities

All the Entities of the Broker of can be managed through the Web Console.

6.2.3.1. Adding Entities

To *add* a new entity, click the `Add` button on the Child Panel on the Parent's tab. Clicking the Add button causes an add dialogue to be displayed.

Add dialogues allow you to set the attributes of the new child, and set context variables. Most fields on the add dialogue have field level help that give more details about the attribute and any default value (which may be expressed in terms of a context variable) that will take effect if you leave the attribute unset. An example add dialogue is shown in the figure that follows.

Figure 6.4. Web Management Orientation - Add Dialogue

6.2.3.2. Editing Entities

To *edit* an existing entity, click the `Edit` button on the tab corresponding to the Entity itself. Editing an entity lets you change some of its attributes and modify its context variables. Most fields on the edit dialogue have field level help that give more details about the attribute and any default value. An example edit dialogue is shown in the figure that follows.

Figure 6.5. Web Management Orientation - Edit Dialogue

6.2.3.3. Deleting Entities

To *remove* an existing entity, click the `Delete` button on the tab corresponding to the Entity itself. For some child types, you can select many children from the parent's type and delete many children at once.

6.2.3.4. Context Variables

All Entities within the Broker have the ability to have context variables associated with them.

Most add and edit dialogues have the ability to make context variable assignments. To add/change/remove a context variable, click the Context Variable panel to expand it.

Figure 6.6. Web Management Orientation - Context Variables

You will see any context variables already associated with the object, called local context variables in bold, and any inherited from above in normal face.

Since context variables can be defined in terms of other context variables, the display has two value columns: actual and effective. Actual shows the value truely associated with the variable, where as effective shows the resulting value, after variable expansion has taken place.

The `+` button allows new variables to be added. The `-` button removes existing ones.

You change an existing local variables defintion by clicking on the actual value. You can also *provide a local definition* for an inherited value by clicking on the actual value and typing its new value.

6.3. REST API

This section provides a brief overview of the REST interfaces, which can be used directly to monitor and manage the Broker instance.

6.3.1. Introduction

The REST interface support traditional REST model which uses the GET method requests to retrieve the information about broker configured objects, DELETE method requests to delete the configured object, PUT to create or update the configured object and POST to perform the configured objects updates and creation.

The REST API is versioned with the version number built into the URL. The general form of the URL is `/api/<version>` where <version> is a major model version prefixed with "v", for example, v3. For convenience the alias `latest` (`/api/latest`) signifies the latest supported version. There are also some ancillary services under URI `/service`.

6.3.2. REST interfaces

The following REST interfaces are implemented on Qpid Broker

Table 6.1. Rest services

REST Service URI	Description
• /api/latest/broker • /api/<version>/broker	Manages broker instance attributes and provides current values of Broker attributes and children
• /api/latest/virtualhostnode • /api/<version>/virtualhostnode • /api/latest/virtualhostnode/<virtual host node name> • /api/<version>/virtualhostnode/<virtual host node name>	Manages(creates/deletes/updates) Virtual Host Node(s) on Broker and provides current values of Virtual Host Node attributes and its children
• /api/latest/virtualhost • /api/<version>/virtualhost • /api/latest/virtualhost/<virtual host node name> • /api/<version>/virtualhost/<virtual host node name> • /api/latest/virtualhost/<virtual host node name>/<virtual host name> • /api/<version>/virtualhost/<virtual host node name>/<virtual host name>	Manages(creates/deletes/updates) Virtual Host on Virtual Host Node and provides current values of Virtual Host attributes and its children
• /api/latest/queue • /api/<version>/queue • /api/latest/queue/<virtual host node name> • /api/<version>/queue/<virtual host node name> • /api/latest/queue/<virtual host node name>/<virtual host name> • /api/<version>/queue/<virtual host node name>/<virtual host name>	Manages(creates/deletes/updates) Queue(s) on Virtual Host and provides current values of Queue attributes and its children

REST Service URI	**Description**
• /api/latest/queue/<virtual host node name>/<virtual host name>/<queue name> • /api/<version>/queue/<virtual host node name>/<virtual host name>/<queue name>	
• /api/latest/exchange • /api/<version>/exchange • /api/latest/exchange/<virtual host node name> • /api/<version>/exchange/<virtual host node name> • /api/latest/exchange/<virtual host node name>/<virtual host name> • /api/<version>/exchange/<virtual host node name>/<virtual host name> • /api/latest/exchange/<virtual host node name>/<virtual host name>/<exchange name> • /api/<version>/exchange/<virtual host node name>/<virtual host name>/<exchange name>	Manages(creates/deletes/updates) Exchanges(s) on Virtual Host and provides current values of Exchange attributes and its children
• /api/latest/binding • /api/<version>/binding • /api/latest/binding/<virtual host node name> • /api/<version>/binding/<virtual host node name> • /api/latest/binding/<virtual host node name>/<virtual host name> • /api/<version>/binding/<virtual host node name>/<virtual host name> • /api/latest/binding/<virtual host node name>/<virtual host name>/<exchange name> • /api/<version>/binding/<virtual host node name>/<virtual host name>/<exchange name> • /api/latest/binding/<virtual host node name>/<virtual host name>/<exchange name>/<queue name>/<binding name> • /api/<version>/binding/<virtual host node name>/<virtual host name>/<exchange name>/<queue name>/<binding name>	Manages(creates/deletes) Binding(s) of Queue to Exchange and provides current values of Binding attributes

REST Service URI	**Description**
• /api/latest/connection • /api/<version>/connection • /api/latest/connection/<virtual host node name> • /api/<version>/connection/<virtual host node name> • /api/latest/connection/<virtual host node name>/<virtual host name> • /api/<version>/connection/<virtual host node name>/<virtual host node name> • /api/latest/connection/<virtual host node name>/<virtual host name>/<connection name> • /api/<version>/connection/<virtual host node name>/<virtual host node name>/<connection name>	Manages (deletes) Connection(s) on VirtualHost and provides current values of Connection attributes and its Sessions
• /api/latest/session • /api/<version>/session • /api/latest/session/<virtual host node name> • /api/<version>/session/<virtual host node name> • /api/latest/session/<virtual host node name>/<virtual host name> • /api/<version>/session/<virtual host node name>/<virtual host node name> • /api/latest/session/<virtual host node name>/<virtual host name>/<connection name> • /api/<version>/session/<virtual host node name>/<virtual host node name>/<connection name • /api/latest/session/<virtual host node name>/<virtual host name>/<connection name>/<session name> • /api/<version>/session/<virtual host node name>/<virtual host node name>/<connection name>/<session name>	Manages (deletes) Session(s) on Connection and provides values of Session attributes and its Consumers
• /api/latest/port • /api/<version>/port • /api/latest/port/<port name>	Manages(creates/deletes/updates) Port(s) on Broker and provides the information about current values of Port attributes and its children

REST Service URI	**Description**
• /api/<version>/port/<port name>	
• /api/latest/authenticationprovider • /api/<version>/authenticationprovider • /api/latest/authenticationprovider/<authentication provider name> • /api/<version>/authenticationprovider/<authentication provider name>	Manages(creates/deletes/updates) AuthenticationProvider(s) on Broker and provides the information about current values of AuthenticationProvider attributes and its children
• /api/latest/user • /api/<version>/user • /api/latest/user/<authentication provider name>/<user name> • /api/<version>/user/<authentication provider name>/<user name> • /api/latest/user/<authentication provider name>/<user name> • /api/<version>/user/<authentication provider name>/<user name>	Manages(creates/deletes/updates) User(s) on AuthenticationProvider and provides the information about current values of User attributes
• /api/latest/accesscontrolprovider • /api/<version>/accesscontrolprovider • /api/latest/accesscontrolprovider/<access control provider name> • /api/<version>/accesscontrolprovider/<access control provider name>	Manages(creates/deletes/updates) AccessControlProvider on Broker and provides the information about current values of AccessControlProvider attributes
• /api/latest/groupprovider • /api/<version>/groupprovider • /api/latest/groupprovider/<group provider name> • /api/<version>/groupprovider/<group provider name>	Manages(creates/deletes/updates) GroupProvider on Broker and provides the information about current values of GroupProvider attributes and its children
• /api/latest/group • /api/<version>/group • /api/latest/group/<group provider name> • /api/<version>/group/<group provider name> • /api/latest/group/<group provider name>/<group name>	Manages(creates/deletes/updates) Group on GroupProvider and provides the information about current values of Group attributes and its children

REST Service URI	**Description**
• /api/<version>/group/<group provider name>/<group name>	
• /api/latest/groupmember • /api/<version>/groupmember • /api/latest/groupmember/<group provider name> • /api/<version>/groupmember/<group provider name> • /api/latest/groupmember/<group provider name>/<group name> • /api/<version>/groupmember/<group provider name>/<group name> • /api/latest/groupmember/<group provider name>/<group name>/<user name> • /api/<version>/groupmember/<group provider name>/<group name>/<user name>	Manages(creates/deletes) GroupMember(s) on Group and provides the information about current values of GroupMember attributes
• /api/latest/keystore • /api/<version>/keystore • /api/latest/keystore/<key store name> • /api/<version>/keystore/<key store name>	Manages(creates/deletes/updates) KeyStore(s) on Broker and provides the information about current values of KeyStore attributes
• /api/latest/truststore • /api/<version>/truststore • /api/latest/truststore/<trust store name> • /api/<version>/truststore/<trust store name>	Manages(creates/deletes/updates) TrustStore(s) on Broker and provides the information about current values of TrustStore attributes
• /api/latest/preferencesprovider • /api/<version>/preferencesprovider • /api/latest/preferencesprovider/<authentication provider name> • /api/<version>/preferencesprovider/<authentication provider name> • /api/latest/preferencesprovider/<authentication provider name>/<preferences provider name> • /api/<version>/preferencesprovider/<authentication provider name>/<preferences provider name>	Manages(creates/deletes/updates) PreferencesProvider on AuthenticationProvider and provides the information about current values of PreferencesProvider attributes

REST Service URI	Description
• /api/latest/plugin • /api/<version>/plugin • /api/latest/plugin/<plugin name> • /api/<version>/plugin/<plugin name>	Manages(creates/deletes/updates) Plugin(s) on Broker and provides the information about current values of Plugin attributes
• /api/latest/remotereplicationnode • /api/<version>/remotereplicationnode • /api/latest/remotereplicationnode/<virtual host node name> • /api/<version>/remotereplicationnode/<virtual host node name> • /api/latest/remotereplicationnode/<virtual host node name>/<replication node name> • /api/<version>/remotereplicationnode/<virtual host node name>/<replication node name>	Manages(creates/deletes/updates) RemoteReplicationNode(s) on VirtualHostNode and provides the information about current values of RemoteReplicationNode attributes
/service/sasl	Sasl authentication. Retrieves user current authentication status and broker supported SASL mechanisms (GET). Authenticates user using supported SASL mechanisms (PUT requests)
/service/logout	Log outs user (GET only)

6.3.3. Configured Object creation

Methods PUT or POST can be used to create ConfiguredObject.

ConfiguredObject can be created by submitting PUT request against ConfiguredObject full URI (the one ending with configured object name) or by submitting PUT/POST request against parent URI. The request encoding should be json (application/json) and request body should contain attributes values in json format. On successful completion of operation a response should be returned having response status code set to 201 and response header "Location" set to ConfiguredObject full URI. If object with a such name/id already exist and POST/PUT requests is made against parent URI, an error response should be returned having response code 409 (conflict) and body containing the json with the reason of operation failure. If object with a such name/id already exist and and PUT request is made against ConfiguredObject full URI, then ConfiguredObject update should be performed and http status code 200 should be returned. If ConfiguredObject cannot be created because of validation failure(s) the response should have http status code set 422 (Unprocessible Entity) and body should contain json with the reason of operation failure. On any other failure to create ConfiguredObject the response should have status code set to 400 (Bad Request) and payload should contain a json with error explaining the exact reason of failure.

Example 6.1. Examples of REST calls for Queue creation

To create Queue with name "my-queue" on a virtual host with name "vh" (which is contained within virtual host node with name "vhn") either of the following requests should be made:

```
PUT /api/latest/queue/vhn/vh HTTP/1.1
```

```
POST /api/latest/queue/vhn/vh HTTP/1.1
```

```
PUT /api/latest/queue/vhn/vh/my-queue HTTP/1.1
```

Response code 201 should be returned on successful queue creation. Response header "Location" should be set to "/api/latest/queue/test/my-queue". If queue with name "my-queue" already exists and either of 2 first requests above were used, an error response with response code 409 (conflict) and body containing json with message that queue exists should be returned. If queue with name "my-queue" exists and last request is used, then Queue update should occur.

6.3.4. Configured Object update

Methods PUT or POST can be used to update ConfiguredObject.

ConfiguredObject can be updated by submitting PUT or POST request against ConfiguredObject full URI (the one ending with configured object name). The request encoding should be json (application/json) and request body should contain a ConfiguredObject json (with all or only modified attributes). On successful completion of operation a response code 200 should be returned. If ConfiguredObject does not exists and PUT method is used, such object should be created (201 response will be returned in this case). If ConfiguredObject does not exists and POST method is used, an error response should be returned having response status code 404 and payload with json explaining the problem. If any error occur on update, a response with response code 400 or 422 or 404 should be sent back to the client containing json body with error details.

Example 6.2. Examples of REST calls for Queue update

To update Queue with name "my-queue" on a virtual host with name "vh" (contained in virtual host node with name "vhn") either of the following requests can be made:

```
POST /api/latest/queue/vhn/vh/my-queue HTTP/1.1
```

```
POST /api/latest/queue/vhn/vh/my-queue HTTP/1.1
```

6.3.5. Configured Object deletion

Method DELETE can be used to delete ConfiguredObject. Alternatively, ConfiguredObject can be deleted with update request having desiredState attribute set to value "DELETED". POST or PUT methods can be used in this case.

On successful completion of operation a response code 200 should be returned.

With DELETE method object ConfiguredObject in following ways:

- by submitting DELETE request using ConfiguredObject full URI (the one ending with configured object name)
- by submitting DELETE request using parent URI and providing parameters having the same names as children attributes, for example, id, name, etc. Multiple children can be deleted in a such way. Many "id" parameters can be specified in such requests. Only children with matching attribute values will be deleted.

Example 6.3. Examples of REST calls for Queue deletion

To delete Queue with name "my-queue" on a virtual host with name "vh" (contained in virtual host node with name "vhn") either of the following requests can be made:

```
DELETE /api/latest/queue/vhn/vh/my-queue HTTP/1.1

DELETE /api/latest/queue/vhn/vh?name=my-queue HTTP/1.1

DELETE /api/latest/queue/vhn/vh?id=real-queue-id HTTP/1.1
```

6.3.6. Retrieving Configured Object details

Method GET is used to retrieve ConfiguredObject attributes values and children hierarchy.

A particular ConfiguredObject details can be retrieved using full ConfiguredObject URI (the one ending with configured object name)

A collection of ConfiguredObjects can be retrieved using parent URI. Request parameters (having the same name as attributes) can be used to filter the returned configured objects.

The REST URI (/api/latest/>category</*) are hierarchical. It is permitted to replace REST URI elements with an "asterisks" in GET requests to denote all object of a particular type. Additionally, trailing object type in the URL hierarchy can be omitted. In this case GET request will return all of the object underneath of the current object.

For example, for binding URL `http://localhost:8080/api/latest/binding/<vhost node>/<vhost>/<exchange>/<queue>/<binding>` replacing of `<exchange>` with "asterisks" (`http://localhost:8080/api/<ver>/binding/<vhost node>/<vhost>/*/<queue>/<binding>`) will result in the GET response containing the list of bindings for all of the exchanges in the virtualhost having the given name and given queue.

If `<binding>` and `<queue>` are omitted in binding REST URL (`http://localhost:8080/api/<ver>/binding/<vhost node>/<vhost>/<exchangename>`) the GET request will result in returning all bindings for all queues for the given exchange in the virtual host.

Additional parameters supported in GET requests:

depth	To restrict the depth of hierarchy of configured objects to return in response
actuals	If set to "true" attribute actual values are returned instead of effective
includeSysContext	If set to "true" all system context variables are returned
inheritedActuals	If set to "true" actual values for all inherited context is returned.
oversize	Sets the maximum length for values of over-sized attributes to trim
extractInitialConfig	If set to "true", the returned json can be used as initial configuration.

6.3.7. HTTP status codes returned by REST interfaces

Table 6.2. HTTP status codes returned by REST interfaces

Status code	Description
200	REST request is successfully completed. This status code can be returned by update, delete and get requests.

Status code	**Description**
201	New configured object is created. It is returned by REST PUT and POST requests for creation of configured objects.
400	REST request cannot be performed due to errors in request. It can be returned from create, update and delete requests. The details of a problem are provided in the response payload in json format.
401	The request requires user authentication
403	Execution of request is not allowed due to failure to authorize user operation.
404	The requested configured object cannot be found. This status code can be returned from POST update requests if configured object does not exist. The reason for the status code is provided in the response payload in json format.
409	The request can not be performed because its execution can create conflicts in the broker. This status code can be returned from POST/PUT create requests against parent URI if configured object with requested name or id already exists. The status code 409 can also be returned if removal or update of configured object can violate system integrity. The reason for the status code is provided in the response payload in json format.
422	The request can not be performed because provided information either incomplete or invalid. This status code can be returned from create or update requests. The reason for the status code is provided in the response payload in json format.

6.3.8. Examples of REST requests with curl

Example 6.4. Examples of queue creation using curl (authenticating as user admin):

```
#create a durable queue
curl --user admin -X PUT  -d '{"durable":true}' http://localhost:8080/api/latest/q
#create a durable priority queue
curl --user admin -X PUT  -d '{"durable":true,"type":"priority"}' http://localhost
```

Example 6.5. Example of binding a queue to an exchange using curl

```
curl --user admin -X PUT  -d '{}' http://localhost:8080/api/latest/binding/<vhostn
```

NOTE: These curl examples utilise unsecure HTTP transport. To use the examples it is first necessary enable Basic authentication for HTTP within the HTTP Management Configuration (it is off by default). For details see Section 7.16, “HTTP Plugin”

6.4. JMX Management

6.4.1. Introduction

The JMX management plugin provides a series of managed beans (MBeans) allowing you to control and monitor the Broker via an industry compliant interface. This provides a convenient integration point for a variety of Infrastructure Monitoring Solutions, tools such as Jconsole and VisualVM, as well as custom Java programs and scripts.

The following sections describe how to connect to JMX, the configuration of the JMX plugin covering topis including securing with SSL, programmatically interacting with Qpid MBeans and finally a summary of all the MBeans made available from by the plugin.

Important

For new development work, the reader is directed towards the strategic Web Management Console and the REST API. Use the Web/REST interfaces in preference to JMX whenever possible. The JMX interface may be withdrawn in a future release.

6.4.2. Default Configuration

By default, the Broker is shipped with JMX enabled.

The RMI registry port runs on port `8999` and the JMX connector on port `9099`. The connector is not SSL protected. Qpid will use the Platform MBeanServer [http://docs.oracle.com/javase/7/docs/api/java/lang/management/ManagementFactory.html#getPlatformMBeanServer()].

To change these settings, use the Web Management interface.

6.4.3. Connecting to JMX

The following example uses Jconsole to illustrates how to connect to JMX and assume the defaults described above. Jconsole is a management tool that comes with the JDK. It provides a very simple view of the MBeans, but requires no special configuration to be used with Qpid.

For full details of Jconsole itself refer to Oracle's JConsole Guide [http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html].

Jconsole can be used to connect to local or remote Java processes. On startup, it presents a list of all the Java processes running on the local host and also allows you to specify a service url to connect to a Java process running on a remote host.

To start Jconsole on the command line, type:

```
jconsole
```

6.4.3.1. Local

To connect to a Broker running locally, simply select the process from the list. You can identify the Broker by looking for its classname `org.apache.qpid.server.Main`.

6.4.3.2. Remote

To connect to a broker running remotely, provide the hostname and port number of the *RMI registry port* (e.g. `hostname:8999`) and a valid username and password.

You can also provide a service url in the form `service:jmx:rmi:///jndi/rmi://hostname:8999/jmxrmi`

Figure 6.7. Making a remote JMX connection to a Broker using jconsole

Once you are connected expand the tree of nodes marked `org.apache.qpid` to begin to interact with the Qpid MBeans.

Figure 6.8. Qpid MBean hierarchy

6.4.3.3. Connecting to a remote Broker protected by SSL

If you are connecting to a remote Broker whose JMX connector port has been secured with SSL certificate signed by a private CA (or a self-signed certificate), you will need to pass a trust store and trust store password to Jconsole. If this is required, start jconsole with the following options:

```
jconsole -J-Djavax.net.ssl.trustStore=jmxtruststore.jks -J-Djavax.net.ssl.trustSto
```

6.4.4. Example JMX Client

The following java snippet illustrates a JMX client that connects to Qpid over JMX passing a userid and password, looks up the ManagedBroker [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ManagedBroker.java?view=co] object corresponding to the `myvhost` virtualhost, then invokes a method on the virtualhost to create a new queue.

A full introduction to custom JMX clients is beyond the scope of this book. For this the reader is directed toward Oracle's JMX tutorial. [http://docs.oracle.com/javase/tutorial/jmx/]

Example 6.6. JMX Client illustrating the creation of a new queue

```
Map<String, Object< environment = new HashMap<String, Object>();
environment.put(JMXConnector.CREDENTIALS, new String[] {"admin","password"});
// Connect to service
JMXServiceURL url =  new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:89
JMXConnector jmxConnector = JMXConnectorFactory.connect(url, environment);
MBeanServerConnection mbsc =  jmxConnector.getMBeanServerConnection();
// Object name for ManagedBroker mbean for virtualhost myvhost
ObjectName objectName = new ObjectName("org.apache.qpid:type=VirtualHost.VirtualHo
// Get the ManagedBroker object
ManagedBroker managedBroker = JMX.newMBeanProxy(mbsc, objectName, ManagedBroker.cl

// Create the queue named "myqueue"
managedBroker.createNewQueue("myqueue", null, true);
```

The Qpid classes required for a custom JMX client are included in the `qpid-management-common` artefact.

6.4.5. The MBeans

The following table summarises the available MBeans. The MBeans are self-describing: each attribute and operation carry a description describing their purpose. This description is visible when using tools such Jconsole. They are also available on Management interfaces themselves (linked below).

Table 6.3. Qpid Broker MBeans

Management Interface	Object Name
ManagedBroker [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ManagedBroker.java?view=co]	`org.apache.qpid:type=VirtualHost.VirtualHostMan` MBean corresponding to the named virtualhost. Allows operations such as the creation/deletion of queues and exchanges on that virtualhost and virtualhost level statistics.
ManagedQueue [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ManagedQueue.java?view=co]	`org.apache.qpid:type=VirtualHost.Queue,VirtualH` MBean corresponding to the named queue on the given virtualhost. Allows queue management operations such as view message, move message and clear queue. Exposes attributes such as queue depth and durability.
ManagedExchange [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ManagedExchange.java?view=co]	`org.apache.qpid:type=VirtualHost.Exchange,Virtu` MBean corresponding to the named exchange on the given virtualhost. Allows exchange management operations such as the creation and removal of bindings. The supported exchange types are exposed by the `exchangeTypes` attribute of the virtualhost.
ManagedConnection [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ManagedConnection.java?view=co]	`org.apache.qpid:type=VirtualHost.Connection,Vir` `*peerid:ephemeralport*"` MBean representing a active AMQP connection to the named virtual host. Name is formed from the IP and ephemeral port of the peer. Attributes include the client version and connection level statistics.
UserManagement [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/UserManagement.java?view=co]	`org.apache.qpid:type=UserManagement,name="UserM` `*authentication manager name*"` When using Plain password provider or Base 64 MD5 password provider, permits user operations such creation and deletion of users. and password changes.
ServerInformation [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/main/java/org/apache/qpid/management/common/mbeans/ServerInformation.java?view=co]	`org.apache.qpid:type=ServerInformation,name=Ser` Exposes broker wide statistics, product version number and JMX management API version number.
LoggingManagement [http://svn.apache.org/viewvc/qpid/java/trunk/management/common/src/	`org.apache.qpid:type=LoggingManagement,name=Log` MBean permitting control of the Broker's logging. Exposes operations allow the logging level to be

Management Interface	**Object Name**
main/java/org/apache/qpid/management/common/mbeans/LoggingManagement.java?view=co]	controlled at runtime (without restarting the Broker) and others that allow changes to be written back to the log4j.xml logging configuration file, or the contents of the log4.xml file to be re-read at runtime.

6.5. AMQP Intrinstic Management

The AMQP protocols 0-8..0-10 allow for creation, deletion and query of Exchanges, Queue and Bindings.

The exact details of how to utilise this commands depends of the client. See the documentation accompanying the client for details.

6.6. QMF

QMF is provided by an optional plugin.

Chapter 7. Managing Entities

This section describes how to manage entities within the Broker. The principles underlying entity management are the same regardless of entity type. For this reason, this section begins with a general description that applies to all.

Since not all channels support the management of all entity type, this section commences with a table showing which entity type is supported by each channel.

7.1. General Description

The following description applies to all entities within the Broker regardless of their type.

- All entities have a parent, and may have children. The parent of the Broker is called the System Context. It has no parent.
- Entities have one or more attributes. For example a `name`, an `id` or a `maximumQueueDepth`
- Entities can be durable or non-durable. Durable entities survive a restart. Non-durable entities will not.
- Attributes may have a default value. If an attribute value is not specified the default value is used.
- Attributes values can be expressed as a simple value (e.g. `myName` or `1234`), in terms of context variables (e.g.`${foo}` or `/data/${foo}/`).
- Each entity has zero or more context variables.
- The System Context entity (the ultimate ancestor of all object) has a context too. It is read only and is populated with all Java System Properties. Thus it can be influenced from the Broker's external environment. See QPID_OPTS environment variable.
- When resolving an attribute's value, if the value contains a variable (e.g.`${foo}`), the variable is first resolved using the entity's own context variables. If the entity has no definition for the context variable, the entity's parent is tried, then its grandparent and so forth, all the way until the SystemContext is reached.
- Some entities support state and have a lifecycle.

What follows now is a section dedicated to each entity type. For each entity type key features are described along with the entities key attributes, key context variables, details of the entities lifecycle and any other operations.

7.2. Entity/Management Channel Support Matrix

This tables indicates which management channels support the creation (C), update (U), or deletion (D) of different entities within the Broker.

Table 7.1. Entity/Management Matrix

Entity	HTTP	JMX	AMQP
Broker	U	No	No
Virtualhost Node	C/U/D	No	No

Entity	**HTTP**	**JMX**	**AMQP**
Virtualhost	C/U/D	No	No
Remote Replication Node	U/D	No	No
Exchange	C/D	C/D	C/D
Queue	C/D	C/U/D	C/D
Binding	C/D	C/D	C/D
Port	C/U/D	No	No
Authentication Providers	C/U/D	No	No
Group Providers	C//D	No	No
Access Control Provider	C//D	No	No
Keystores	C//D	No	No
Truststores	C//D	No	No
Users	C//D	C/D	No
Groups	C//D	No	No

Note

It is currently only possible to modify a entity's context using the HTTP channel. Furthermore, when using the JMX channel, it is not possible to assign non-String type attributes in terms of context variables.

7.3. Broker

The Broker is the principal entity. It is composed of a number of other entities that collaborate to provide message broker facilities.

The Broker can only be managed via the HTTP management channel.

7.3.1. Attributes

- *Name the Broker*. This helps distinguish between Brokers in environments that have many.
- *Heartbeating*. Enables heartbeats between Broker and Clients. Heartbeats help discover severed TCP/IP connections in a timely manner.
- *Confidential configuration encryption provider*. The name of the provider used to encrypt passwords and other secrets within the configuration. See Section 8.4, “Configuration Encryption”.

7.3.2. Context

- *broker.flowToDiskThreshold* Controls the flow to disk feature.
- *broker.messageCompressionEnabled* Controls the message compression .

7.3.3. Children

- Virtualhost nodes

- Ports
- Authentication Providers
- Key Stores / Trust Stores
- Group Providers
- Access Control Providers

7.3.4. Lifecycle

Not supported

7.4. Virtualhost Nodes

Virtualhost nodes can only be managed by the HTTP management channel.

7.4.1. Types

The following virtualhost nodes types are supported.

- BDB - Node backed with Oracle BDB [1]
- BDB HA - Node backed with Oracle BDB utilising High Availability
- DERBY - Node backed with Apache Derby
- JDBC - Node backed with an external database [2]
- JSON - Node backed with a file containing json
- Memory - In-memory node (changes lost on Broker restart)

7.4.2. Attributes

- *Name the virtualhost node*.
- *Default Virtual Host Node*. If true, messaging clients which do not specify a virtualhost name will be connected to the virtualhost beneath this node.
- *Store Path or JDBC URL.* Refers the location used to store the configuration of the virtualhost.
- *Role* (HA only). The role that this node is currently playing in the group.
  - MASTER - Virtualhost node is a master.
  - REPLICA - Virtualhost node is a replica.
  - WAITING - Virtualhost node is awaiting an election result, or may be awaiting more nodes to join in order that an election may be held.

[1]Oracle BDB JE is optional. See Section E.3, “Installing Oracle BDB JE”.
[2]JDBC 4.0 compatible drivers must be available. See Section E.2, “Installing External JDBC Driver”

  - DETACHED - Virtualhost node is disconnected from the group.

- *Priority* (HA only). The priority of this node when elections occurs. The attribute can be used to make it more likely for a node to be elected than other nodes, or disallow the node from never being elected at all. See Section 10.4.3, “Node Priority”
- *Minimum Number Of Nodes* (HA only - groups of three or more). Allows the number of nodes required to hold an election to be reduced in order that service can be restore when less than quorum nodes are present. See Section 10.4.4, “Required Minimum Number Of Nodes”
- *Designated Primary* (HA only - groups of two). Allows a single node in a two node group to operate solo. See Section 10.4.5, “Designated Primary”

7.4.3. Children

- Virtualhost
- Remote Replication Nodes

7.4.4. Lifecycle

- *Stop*. Stops the virtualhost node. This closes any existing messaging connections to the virtualhost and prevents new ones. Any inflight transactions are rolled back. Non durable queues and exchanges are lost. Transient messages or persistent messages on non-durable queues are lost.

  When HA is in use, stopping the virtualhost node stops the virtualhost node from participating in the group. If the node was in the master role, the remaining nodes will try to conduct an election and elect a new master. If the node was in the replica role, the node will cease to keep up to date with later transactions. A stopped node does not vote in elections. Other nodes in the group will report the stopped node as unreachable.
- *Start*. Activates the virtualhost node.
- *Delete*. Deletes the virtualhost node and the virtualhost contained within it. All exchanges and queues, any the messages contained within it are removed. In the HA case, deleting the virtualhost node causes it be removed permanently from the group.

7.5. VirtualHosts

A virtualhost is a independent namespace in which messaging is performed. Virtualhosts are responsible for the storage of message data.

Virtualhosts can only be managed by the HTTP management channel.

7.5.1. Types

The following virtualhost types are supported.

- BDB - Virtualhost backed with Oracle BDB [3]
- BDB HA - Virtualhost backed with Oracle BDB utilising High Availability

[3]Oracle BDB JE is optional. See Section E.3, “Installing Oracle BDB JE”.

- DERBY - Virtualhost backed with Apache Derby
- JDBC - Virtualhost backed with an external database [4]
- Memory - In-memory node (changes lost on Broker restart)
- Provided - Virtualhost that co-locates message data within the parent virtualhost node [5].

7.5.2. Context

- *use_async_message_store_recovery* Controls the background recovery feature.

7.5.3. Attributes

- *Name the virtualhost*. This is the name the messaging clients refer to when forming a connection to the Broker.
- *Store Path/JDBC URL*. Refers the file system location or database URL used to store the message data.
- *Store overflow/underflow*. Some virtualhosts have the ability to limit the of the cumulative size of all the messages contained within the store. This feature is described in detail Section 9.2, “Disk Space Management”.
- *Store transaction timeouts*. Warns of long running producer transactions. See Section 9.3, “Producer Transaction Timeout”
- *Synchronization policy*. HA only. See Section 10.4.2, “Synchronization Policy”

7.5.4. Children

- Exchange
- Queue
- Connection

7.5.5. Lifecycle

- *Stop*. Stops the virtualhost. This closes any existing messaging connections to the virtualhost and prevents new ones. Any inflight transactions are rolled back. Non durable queues and non durable exchanges are lost. Transient messages or persistent messages on non-durable queues are lost.
- *Start*. Activates the virtualhost.

7.6. Remote Replication Nodes

Used for HA only. A remote replication node is a representation of another virtualhost node in the group. Remote replication nodes are not created directly. Instead the system automatically creates a remote replication node for every node in the group. It serves to provide a view of the whole group from every node in the system.

[4]JDBC 4.0 compatible drivers must be available. See Section E.2, “Installing External JDBC Driver”
[5]Not available if Virtualhost Node type is JSON.

7.6.1. Attributes

- *Name the remote replication node*. This is the name of the remote virtualhost node
- *Role*. Indicates the role that the remote node is playing in the group at this moment.
  - *MASTER* - Remote node is a master.
  - *REPLICA* - Remote node is a replica.
  - *UNREACHABLE* - Remote node unreachable from this node. This remote note may be down, or an network problem may prevent it from being contacted.
- *Join time*. Time when first contact was established with this node.
- *Last known transaction id*. Last transaction id reported processed by node. This is an internal transaction counter and does not relate to any value available to the messaging clients. This value can only be used to determine the node is up to date relative to others in the group.

7.6.2. Children

None

7.6.3. Lifecycle

- *Delete*. Causes the remote node to be permanently removed from the group. This operation should be used when the virtualhost node cannot be deleted from its own Broker, for instance, if a Broker has been destroyed by machine failure.

7.6.4. Operations

- *Transfer Master*. Initiates a process where a master is moved to anther node in the group. The transfer sequence is as follows.
  1. Group waits until the proposed master is reasonable up to date.
  2. Any in-flight transactions on the current master are blocked.
  3. The current master awaits the proposed master to become up to date.
  4. The mastership is transferred. This will automatically disconnect messaging clients from the old master, and in-flight transactions are rolled back. Messaging clients reconnect to the new master.
  5. The old master will rejoin as a replica.

7.7. Exchanges

Exchanges can be managed using the HTTP, JMX or AMQP channels.

7.7.1. Types

- Direct
- Topic

- Fanout
- Headers

7.7.2. Attributes

- *Name of the exchange*. Message producers refer to this name when producing messages.
- *Type of the exchange*. Can be either direct, topic, fanout, or headers.
- *Durable*. Whether the exchange survives a restart.

7.7.3. Children

- Binding

7.7.4. Lifecycle

Not supported

7.8. Queues

Queues are named entities that hold/buffer messages for later delivery to consumer applications.

Queues can be managed using the HTTP, JMX or AMQP channels.

7.8.1. Types

The Broker supports four different queue types, each with different delivery semantics.

- Standard - a simple First-In-First-Out (FIFO) queue
- Priority - delivery order depends on the priority of each message
- Sorted - delivery order depends on the value of the sorting key property in each message
- Last Value Queue - also known as an LVQ, retains only the last (newest) message received with a given LVQ key value

7.8.2. Attributes

- *Name of the queue*. Message consumers and browsers refer to this name when they wish to subscribe to queue to receive messages from it.
- *Type of the queue*. Can be either standard, priority, sorted, or lvq.
- *Durable*. Whether the queue survives a restart. Messages on a non durable queue do not survive a restart even if they are marked persistent.
- *Maximum/Minimum TTL*. Defines a maximum and minimum time-to-live. Messages arriving with ttl larger than the maximum will be overridden by the maximum. Similarly, messages arriving with tll less than the minimum (or no ttl at all), will be overridden by the minimum.

Changing these values affects only new arrivals, existing messages already on the queue are not affected.

- *Message persistent override*. Allow message persistent settings of incoming messages to be overridden. Changing this value affects only new arrivals, existing messages on the queue are not affected.
- *Queue capacity*. Queues have the ability to limit the of the cumulative size of all the messages contained within the store. This feature is described in detail Section 9.2, “Disk Space Management”.
- *Alerting Thresholds*. Queues have the ability to alert on a variety of conditions: total queue depth exceeded a number or size, message age exceeded a threshold, message size exceeded a threshold. These thresholds are soft. See Appendix D, *Queue Alerts*
- *Maximum Delivery Count/Alternate Exchange*. See Section 9.4, “Handing Undeliverable Messages”
- *Message Groups*. See Section 4.7.3, “Messaging Grouping”

7.8.3. Children

- Binding

7.8.4. Lifecycle

Not supported

7.9. Consumers

A Consumer represents an application's live *subcription* to a queue. Its presence in the model indicates that an application is currently connected to the queue *at this moment*.

7.9.1. Context

- *consumer.suspendNotificationPeriod* Governs the length of time that a consumer may remain suspended before the the Broker begins to produce SUB-1003 operational log messages.

7.10. Ports

Ports provide TCP/IP connectivity for messaging and management. A port is defined to use a protocol. This can be an AMQP protocol for messaging or HTTP/JMX for management.

A port is defined to have one or more transports. A transport can either be plain (TCP) or SSL. When SSL is in use, the port can be configured to accept or require client authentication.

Any number of ports defined to use AMQP or HTTP protocols can be defined. JMX is limited to a single port instance per JMX protocol type.

Ports can only be managed by the HTTP management channel.

7.10.1. Attributes

- *Name the port*.
- *Port number*.

- *Binding address*. Used to limit port binding to a single network interface.
- *Authentication Provider*. The authentication provider used to authenticate incoming connections.
- *Protocol(s)*. A list of protocols to be supported by the port. For messaging choose one or more AMQP protocols. For management choose HTTP or on one the two JMX protocols.
- *Transports*. A list of transports supported by the port. For messaging or HTTP management chose TCP, SSL or both. For JMX, the TCP/SSL combination is not supported.
- *Keystore*. Keystore containing the Broker's private key. Required if the SSL is in use.
- *Want/Need Client Auth*. Client authentication can be either accepted if offered (want), or demanded (need). When Client Certificate Authentication is in use a Truststore must be configured. When using Client Certificate Authentication it may be desirable to use the External Authentication Provider.

  JMX does not support client authentication.
- *Truststore*. Trust store contain an issuer certificate or the public keys of the clients themselves if peers only is desired.

7.10.2. Children

None

7.10.3. Lifecycle

Not supported

Important

When updating an existing port, changes to SSL settings, binding address and port numbers do not become effective until the Broker is restarted.

7.11. Authentication Providers

Authentication Providers are used by Ports to authenticate connections.

See Section 8.1, “Authentication Providers”

7.11.1. Types

The following authentication providers are supported:

- Anonymous: allows anonymous connections to the Broker
- External: delegates to external mechanisms such as SSL Client Certificate Authentication
- Kerberos: uses Kerberos to authenticate connections via GSS-API.
- SimpleLDAP: authenticate users against an LDAP server.
- ScramSha: authenticate users against credentials stored in a local database
- Plain: authenticate users against credentials stored in a local database.

- PlainPasswordFile: authenticate users against credentials stored in plain text in a local file.
- MD5: authenticate users against credentials stored in a local database.
- Base64MD5PasswordFile: authenticate users against credentials stored encoded in a local file.

The last five providers offer user management facilities too, that is, users can be created, deleted and passwords reset.

7.11.2. Attributes

- *Name the authentication provider*.

Other attributes are provider specific.

7.11.3. Children

None

7.11.4. Lifecycle

Not supported

Important

When updating an existing authentication provider, changes become effective until the Broker is restarted.

7.12. Keystores

A Keystore is required by a Port in order to use SSL.

7.12.1. Attributes

- *Name the keystore*. Used to identify the keystore.
- *Path*. Path to keystore file
- *Keystore password*. Password used to secure the keystore

  Important

  The password of the certificate used by the Broker **must** match the password of the keystore itself. This is a restriction of the Qpid Broker implementation. If using the keytool [http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html] utility, note that this means the argument to the `-keypass` option must match the `-storepass` option.

- *Certificate Alias*. An optional way of specifying which certificate the broker should use if the keystore contains multiple entries.
- *Manager Factory Algorithm*. In keystores the have more than one certificate, the alias identifies the certificate to be used.

- *Key Store Type*. Type of Keystore.

7.12.2. Children

None

7.12.3. Lifecycle

Not supported

7.13. Truststores

A Truststore is required by a Port in order to SSL client authentication. Some authentication provides also use a truststore when connecting to authentication systems that are protected by a private issuer SSL certificate.

7.13.1. Attributes

- *Name the truststore*. Used to identify the truststore.
- *Path*. Path to truststore file
- *Truststore password*. Password used to secure the truststore

  Important

  The password of the certificate used by the Broker **must** match the password of the keystore itself.

- *Certificate Alias*. An optional way of specifying which certificate the broker should use if the keystore contains multiple entries.
- *Manager Factory Algorithm*. In keystores the have more than one certificate, the alias identifies the certificate to be used.
- *Key Store Type*. Type of Keystore.
- *Peers only*. When "Peers Only" option is selected for the Truststore it will allow authenticate only those clients that present a certificate exactly matching a certificate contained within the Truststore database.

7.13.2. Children

None

7.13.3. Lifecycle

Not supported

7.14. Group Providers

See Section 8.2, “Group Providers”

7.15. Access Control Providers

An Access Control Provider governs who may do what within the Broker. It governs both messaging and management.

See Section 8.3, “Access Control Lists”

7.16. HTTP Plugin

The HTTP Plugin provides the HTTP management channel comprising of the Web Management Console and the REST API.

7.16.1. Attributes

- *Basic Authentication for HTTP*. It is set to false (disabled) by default.
- *Basic Authentication for HTTPS*. It is set to true (enabled) by default.
- *SASL Authentication for HTTP*. It is set to true (enabled) by default.
- *SASL Authentication for HTTPS*. It is set to true (enabled) by default.
- *Session timeout* is the timeout in seconds to close the HTTP session. It is set to 10 minutes by default.

7.16.2. Children

None

7.16.3. Lifecycle

Not supported

Important

NOTE: Changes to the Session Timeout attribute only take effect at broker restart.

7.17. JMX Plugin

The JMX Plugin provides the JMX management channel.

TODO

Chapter 8. Security

8.1. Authentication Providers

In order to successfully establish a connection to the Java Broker, the connection must be authenticated. The Java Broker supports a number of different authentication schemes, each with its own "authentication provider". Any number of Authentication Providers can be configured on the Broker at the same time.

Important

Only unused Authentication Provider can be deleted. For delete requests attempting to delete Authentication Provider associated with the Ports, the errors will be returned and delete operations will be aborted. It is possible to change the Authentication Provider on Port at runtime. However, the Broker restart is required for changes on Port to take effect.

Note

Authentication Providers may choose to selectively disable certain authentication mechanisms depending on whether an encrypted transport is being used or not. This is to avoid insecure configurations. Notably, by default the PLAIN mechanism will be disabled on non-SSL connections. This security feature can be overwritten by setting

```
secureOnlyMechanisms = []
```

in the authentication provider section of the config.json.

Warning

Changing the secureOnlyMechanism is a breach of security and might cause passwords to be transfered in the clear. Use at your own risk!

8.1.1. Simple LDAP

The Simple LDAP authenticates connections against a Directory (LDAP).

To create a SimpleLDAPAuthenticationProvider the following mandatory fields are required:

- *LDAP server URL* is the URL of the server, for example, `ldaps://example.com:636`
- *Search context* is the distinguished name of the search base object. It defines the location from which the search for users begins, for example, `dc=users,dc=example,dc=com`
- *Search filter* is a DN template to find an LDAP user entry by provided user name, for example, `(uid={0})`

Additionally, the following optional fields can be specified:

- *LDAP context factory* is a fully qualified class name for the JNDI LDAP context factory. This class must implement the InitialContextFactory [http://docs.oracle.com/javase/7/docs/api/javax/naming/spi/InitialContextFactory.html] interface and produce instances of DirContext [http://docs.oracle.com/javase/7/docs/api/javax/naming/directory/DirContext.html]. If not specified a default value of `com.sun.jndi.ldap.LdapCtxFactory` is used.

- *LDAP authentication URL* is the URL of LDAP server for performing "ldap bind". If not specified, the *LDAP server URL* will be used for both searches and authentications.
- *Truststore name* is a name of configured truststore. Use this if connecting to a Directory over SSL (i.e. ldaps://) which is protected by a certificate signed by a private CA (or utilising a self-signed certificate).

Important

In order to protect the security of the user's password, when using LDAP authentication, you must:

- Use SSL on the broker's AMQP, HTTP and JMX ports to protect the password during transmission to the Broker. The Broker enforces this restriction automatically on AMQP and HTTP ports.
- Authenticate to the Directory using SSL (i.e. ldaps://) to protect the password during transmission from the Broker to the Directory.

The LDAP Authentication Provider works in the following manner. If not in `bind without search` mode, it first connects to the Directory and searches for the ldap entity which is identified by the username. The search begins at the distinguished name identified by `Search Context` and uses the username as a filter. The search scope is sub-tree meaning the search will include the base object and the subtree extending beneath it.

If the search returns a match, or is configured in `bind without search` mode, the Authentication Provider then attempts to bind to the LDAP server with the given name and the password. Note that simple security authentication [http://docs.oracle.com/javase/7/docs/api/javax/naming/Context.html#SECURITY_AUTHENTICATION] is used so the Directory receives the password in the clear.

8.1.2. Kerberos

Kereberos Authentication Provider uses java GSS-API SASL mechanism to authenticate the connections.

Configuration of kerberos is done through system properties (there doesn't seem to be a way around this unfortunately).

```
    export JAVA_OPTS=-Djavax.security.auth.useSubjectCredsOnly=false -Djava.securi
    ${QPID_HOME}/bin/qpid-server
```

Where qpid.conf would look something like this:

```
com.sun.security.jgss.accept {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    doNotPrompt=true
    realm="EXAMPLE.COM"
    useSubjectCredsOnly=false
    kdc="kerberos.example.com"
    keyTab="/path/to/keytab-file"
    principal="<name>/<host>";
};
```

Where realm, kdc, keyTab and principal should obviously be set correctly for the environment where you are running (see the existing documentation for the C++ broker about creating a keytab file).

Note: You may need to install the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" appropriate for your JDK in order to get Kerberos support working.

Since Kerberos support only works where SASL authentication is available (e.g. not for JMX authentication) you may wish to also include an alternative Authentication Provider configuration, and use this for JMX and HTTP ports.

8.1.3. External (SSL Client Certificates)

When requiring SSL Client Certificates be presented the External Authentication Provider can be used, such that the user is authenticated based on trust of their certificate alone, and the X500Principal from the SSL session is then used as the username for the connection, instead of also requiring the user to present a valid username and password.

Note: The External Authentication Provider should typically only be used on the AMQP/HTTP ports, in conjunction with SSL client certificate authentication. It is not intended for other uses such as the JMX management port and will treat any non-sasl authentication processes on these ports as successful with the given username. As such you should configure another Authentication Provider for use on JMX ports.

On creation of External Provider the use of full DN or username CN as a principal name can be configured. If attribute "Use the full DN as the Username" is set to "true" the full DN is used as an authenticated principal name. If attribute "Use the full DN as the Username" is set to "false" the user name CN part is used as the authenticated principal name. Setting the field to "false" is particular useful when ACL is required, as at the moment, ACL does not support commas in the user name.

8.1.4. Anonymous

The Anonymous Authentication Provider will allow users to connect with or without credentials and result in their identification on the broker as the user ANONYMOUS. This Provider does not require specification of any additional attributes on creation.

8.1.5. SCRAM SHA

The SCRAM SHA Providers uses the Broker configuration itself to store the database of users. The users' passwords are stored as salted SHA digested password. This can be further encrypted using the facilities described in Section 8.4, “Configuration Encryption”.

There are two variants of this provider, SHA1 and SHA256. SHA256 is recommended whenever possible. SHA1 is provided with compatibility with clients utilising JDK 1.6 (which does not support SHA256).

For these providers user credentials can be added, removed or changed using Management.

8.1.6. Plain

The Plain Provider uses the Broker configuration itself to store the database of users (unlike the PlainPasswordFile, there is no separate password file). As the name suggests, the user data (including password) is not hashed in any way. In order to provide encryption, the facilities described in Section 8.4, “Configuration Encryption” must be used.

For this provider user credentials can be added, removed or changed using Management.

8.1.7. Plain Password File *(Deprecated)*

This provider is deprecated and will be removed in a future release. The Plain provider should be used instead.

The PlainPasswordFile Provider uses local file to store and manage user credentials. When creating an authentication provider the path to the file needs to be specified. If specified file does not exist an empty file is created automatically on Authentication Provider creation. On Provider deletion the password file is deleted as well.

For this provider user credentials can be added, removed or changed using Management.

8.1.7.1. Plain Password File Format

The user credentials are stored on the single file line as user name and user password pairs separated by colon character. This file must not be modified externally whilst the Broker is running.

```
# password file format
# <user name>: <user password>
guest:guest
```

8.1.8. MD5 Provider

MD5 Provider uses the Broker configuration itself to store the database of users (unlike the Base64MD5 Password File, there is no separate password file). Rather than store the unencrypted user password (as the Plain provider does) it instead stores the MD5 password digest. This can be further encrypted using the facilities described in Section 8.4, “Configuration Encryption”.

For this provider user credentials can be added, removed or changed using Management.

8.1.9. Base64MD5 Password File *(Deprecated)*

This provider is deprecated and will be removed in a future release. The MD5 provider should be used instead.

Base64MD5PasswordFile Provider uses local file to store and manage user credentials similar to PlainPasswordFile but instead of storing a password the MD5 password digest encoded with Base64 encoding is stored in the file. When creating an authentication provider the path to the file needs to be specified. If specified file does not exist an empty file is created automatically on Authentication Provider creation. On Base64MD5PasswordFile Provider deletion the password file is deleted as well.

For this provider user credentials can be added, removed or changed using Management.

8.1.9.1. Base64MD5 File Format

The user credentials are stored on the single file line as user name and user password pairs separated by colon character. The password is stored MD5 digest/Base64 encoded. This file must not be modified externally whilst the Broker is running.

8.2. Group Providers

The Java broker utilises GroupProviders to allow assigning users to groups for use in ACLs. Following authentication by a given Authentication Provider, the configured Group Providers are consulted allowing the assignment of GroupPrincipals for a given authenticated user. Any number of Group Providers can be added into the Broker. All of them will be checked for the presence of the groups for a given authenticated user.

8.2.1. GroupFile Provider

The *GroupFile* Provider allows specifying group membership in a flat file on disk. On adding a new GroupFile Provider the path to the groups file is required to be specified. If file does not exist an empty file is created automatically. On deletion of GroupFile Provider the groups file is deleted as well. Only one instance of "GroupFile" Provider per groups file location can be created. On attempt to create another GroupFile Provider pointing to the same location the error will be displayed and the creation will be aborted.

8.2.1.1. File Format

The groups file has the following format:

```
# <GroupName>.users = <comma delimited user list>
# For example:

administrators.users = admin,manager
```

Only users can be added to a group currently, not other groups. Usernames can't contain commas.

Lines starting with a '#' are treated as comments when opening the file, but these are not preserved when the broker updates the file due to changes made through the management interface.

8.3. Access Control Lists

In Qpid, Access Control Lists (ACLs) specify which actions can be performed by each authenticated user. To enable, an *Access Control Provider* needs to be configured on the *Broker*. The *Access Control Provider* of type "AclFile" uses local file to specify the ACL rules. By convention, this file should have a .acl extension.

A Group Provider can be configured with ACL to define the user groups which can be used in ACL to determine the ACL rules applicable to the entire group. The configuration details for the Group Providers are described in Section 8.2, “Group Providers”. On creation of ACL Provider with group rules, the Group Provider should be added first. Otherwise, if the individual ACL rules are not defined for the logged principal the following invocation of management operations could be denied due to absence of the required groups.

Only one *Access Control Provider* can be used by the Broker. If several *Access Control Providers* are configured on Broker level only one of them will be used (the latest one).

The ACL Providers can be configured using REST Management interfaces and Web Management Console.

The following ACL Provider managing operations are available from Web Management Console:

- A new ACL Provider can be added by clicking onto "Add Access Control Provider" on the Broker tab.
- An ACL Provider details can be viewed on the Access Control Provider tab. The tab is shown after clicking onto ACL Provider name in the Broker object tree or after clicking onto ACL Provider row in ACL Providers grid on the Broker tab.

- An existing ACL Provider can be deleted by clicking onto buttons "Delete Access Control Provider" on the Broker tab or Access Control Provider tab.

8.3.1. Writing .acl files

The ACL file consists of a series of rules associating behaviour for a user or group. Use of groups can serve to make the ACL file more concise. See Configuring Group Providers for more information on defining groups.

Each ACL rule grants or denies a particular action on an object to a user/group. The rule may be augmented with one or more properties, restricting the rule's applicability.

```
      ACL ALLOW alice CREATE QUEUE              # Grants alice permission to creat
      ACL DENY bob CREATE QUEUE name="myqueue"  # Denies bob permission to create
```

The ACL is considered in strict line order with the first matching rule taking precedence over all those that follow. In the following example, if the user bob tries to create an exchange "myexch", the operation will be allowed by the first rule. The second rule will never be considered.

```
      ACL ALLOW bob ALL EXCHANGE
      ACL DENY bob CREATE EXCHANGE name="myexch"  # Dead rule
```

If the desire is to allow bob to create all exchanges except "myexch", order of the rules must be reversed:

```
      ACL DENY bob CREATE EXCHANGE name="myexch"
      ACL ALLOW bob ALL EXCHANGE
```

All ACL files end with an implicit rule denying all operations to all users. It is as if each file ends with

```
ACL DENY ALL ALL
```

If instead you wish to *allow* all operations other than those controlled by earlier rules, add

```
ACL ALLOW ALL ALL
```

to the bottom of the ACL file.

When writing a new ACL, a good approach is to begin with an .acl file containing only

```
ACL DENY-LOG ALL ALL
```

which will cause the Broker to deny all operations with details of the denial logged to the Qpid log file. Build up the ACL rule by rule, gradually working through the use-cases of your system. Once the ACL is complete, consider switching the DENY-LOG actions to DENY to improve performamce and reduce log noise.

ACL rules are very powerful: it is possible to write very granular rules specifying many broker objects and their properties. Most projects probably won't need this degree of flexibility. A reasonable approach is to choose to apply permissions at a certain level of abstraction (e.g. QUEUE) and apply them consistently across the whole system.

Note

Some rules can be restricted to the virtual host if property virtualhost_name is specified.

Example 8.1. Restrict rules to specific virtual hosts

```
ACL ALLOW bob CREATE QUEUE virtualhost_name="test"
ACL ALLOW bob ALL EXCHANGE virtualhost_name="prod"
```

In the example above the first rule allows user "bob" to create queues on virtual host "test" only. The second rule allows user "bob" any action with exchanges on virtual host "prod".

8.3.2. Syntax

ACL rules follow this syntax:

```
ACL {permission} {<group-name>|<user-name>|ALL} {action|ALL} [object|ALL] [pr
```

Comments may be introduced with the hash (#) character and are ignored. Long lines can be broken with the slash (\) character.

```
# A comment
ACL ALLOW admin CREATE ALL # Also a comment
ACL DENY guest \
ALL ALL   # A broken line
```

Table 8.1. List of ACL permission

ALLOW	Allow the action
ALLOW-LOG	Allow the action and log the action in the log
DENY	Deny the action
DENY-LOG	Deny the action and log the action in the log

Table 8.2. List of ACL actions

Action	Description	Supported object types	Supported properties
CONSUME	Applied when subscriptions are created	QUEUE	name, autodelete, temporary, durable, exclusive, alternate, owner, virtualhost_name
PUBLISH	Applied on a per message basis on publish message transfers	EXCHANGE	name, routingkey, immediate, virtualhost_name
CREATE	Applied when an object is created, such as bindings, queues, exchanges	VIRTUALHOSTNODE, VIRTUALHOST, EXCHANGE, QUEUE, USER, GROUP	see properties on the corresponding object type

Action	Description	Supported object types	Supported properties
ACCESS	Applied when an object is read or accessed	VIRTUALHOST, MANAGEMENT	name (for VIRTUALHOST only)
BIND	Applied when queues are bound to exchanges	EXCHANGE	name, routingKey, queuename, virtualhost_name, temporary, durable
UNBIND	Applied when queues are unbound from exchanges	EXCHANGE	name, routingKey, queuename, virtualhost_name, temporary, durable
DELETE	Applied when objects are deleted	VIRTUALHOSTNODE, VIRTUALHOST, EXCHANGE, QUEUE, USER, GROUP	see properties on the corresponding object type
PURGE	Applied when purge the contents of a queue	QUEUE	
UPDATE	Applied when an object is updated	VIRTUALHOSTNODE, VIRTUALHOST, EXCHANGE, QUEUE, USER, GROUP	see EXCHANGE and QUEUE properties
CONFIGURE	Applied when an object is configured via REST management interfaces.	BROKER	
ACCESS_LOGS	Allows/denies to the specific user an operation to download broker log file(s) over REST interfaces	BROKER	

Table 8.3. List of ACL objects

Object type	Description	Supported actions	Supported properties
VIRTUALHOSTNODE	A virtualhostnode or remote replication node	ALL, CREATE, UPDATE, DELETE	name
VIRTUALHOST	A virtualhost	ALL, CREATE, UPDATE, DELETE, ACCESS	name
MANAGEMENT	Management - for web and JMX	ALL, ACCESS	
QUEUE	A queue	ALL, CREATE, DELETE, PURGE, CONSUME, UPDATE	name, autodelete, temporary, durable, exclusive, alternate, owner, virtualhost_name
EXCHANGE	An exchange	ALL, ACCESS, CREATE, DELETE, BIND, UNBIND, PUBLISH, UPDATE	name, autodelete, temporary, durable, type, virtualhost_name, queuename(only for BIND and UNBIND),

Object type	Description	Supported actions	Supported properties
			routingkey(only for BIND and UNBIND, PUBLISH)
USER	A user	ALL, CREATE, DELETE, UPDATE	name
GROUP	A group	ALL, CREATE, DELETE, UPDATE	name
METHOD	Management or agent or broker method	ALL, ACCESS, UPDATE	name, component, virtualhost_name
BROKER	The broker	ALL, CONFIGURE, ACCESS_LOGS	

Table 8.4. List of ACL properties

name	String. Object name, such as a queue name, exchange name or JMX method name.
durable	Boolean. Indicates the object is durable
routingkey	String. Specifies routing key
autodelete	Boolean. Indicates whether or not the object gets deleted when the connection is closed
exclusive	Boolean. Indicates the presence of an `exclusive` flag
temporary	Boolean. Indicates the presence of an `temporary` flag
type	String. Type of object, such as topic, fanout, or xml
alternate	String. Name of the alternate exchange
queuename	String. Name of the queue (used only when the object is something other than `queue`
component	String. JMX component name
from_network	Comma-separated strings representing IPv4 address ranges. Intended for use in ACCESS VIRTUALHOST rules to apply firewall-like restrictions. The rule matches if any of the address ranges match the IPv4 address of the messaging client. The address ranges are specified using either Classless Inter-Domain Routing notation (e.g. 192.168.1.0/24; see RFC 4632 [http://tools.ietf.org/html/rfc4632]) or wildcards (e.g. 192.169.1.*).
from_hostname	Comma-separated strings representing hostnames, specified using Perl-style regular expressions, e.g. .*\.example\.company\.com Intended for use in ACCESS VIRTUALHOST rules to apply firewall-like restrictions.

	The rule matches if any of the patterns match the hostname of the messaging client. To look up the client's hostname, Qpid uses Java's DNS support, which internally caches its results. You can modify the time-to-live of cached results using the *.ttl properties described on the Java Networking Properties [http://docs.oracle.com/javase/6/docs/technotes/guides/net/properties.html] page. For example, you can either set system property sun.net.inetaddr.ttl from the command line (e.g. export QPID_OPTS="-Dsun.net.inetaddr.ttl=0") or networkaddress.cache.ttl in $JAVA_HOME/lib/security/java.security. The latter is preferred because it is JVM vendor-independent.
virtualhost_name	String. A name of virtual host to which the rule is applied.
immediate	Boolean. A property can be used to restrict PUBLISH action to publishing only messages with given immediate flag.

Table 8.5. List of ACL JMX Components

UserManagement	User maintenance; create/delete/view users, change passwords etc
ConfigurationManagement	Dynamically reload configuration from disk.
LoggingManagement	Dynamically control Qpid logging level
ServerInformation	Read-only information regarding the Qpid: version number etc
VirtualHost.Queue	Queue maintenance; copy/move/purge/view etc
VirtualHost.Exchange	Exchange maintenance; bind/unbind queues to exchanges
VirtualHost.VirtualHost	Virtual host maintenace; create/delete exchanges, queues etc

8.3.3. Worked Examples

Here are some example ACLs illustrating common use cases. In addition, note that the Java broker provides a complete example ACL file, located at etc/broker_example.acl.

8.3.3.1. Worked example 1 - Management rights

Suppose you wish to permission two users: a user 'operator' must be able to perform all Management operations, and a user 'readonly' must be enable to perform only read-only functions. Neither 'operator' nor 'readonly' should be allowed to connect clients for messaging.

```
# Deny (loggged) operator/readonly permission to connect messaging clients.
```

```
ACL DENY-LOG operator ACCESS VIRTUALHOST
ACL DENY-LOG readonly ACCESS VIRTUALHOST
# Give operator permission to perfom all other actions
ACL ALLOW operator ALL ALL
# Give readonly permission to execute only read-only actions
ACL ALLOW readonly ACCESS ALL
...
... rules for other users
...
# Explicitly deny all (log) to eveyone
ACL DENY-LOG ALL ALL
```

8.3.3.2. Worked example 2 - User maintainer group

Suppose you wish to restrict User Management operations to users belonging to a group 'usermaint'. No other user is allowed to perform user maintenance This example illustrates the permissioning of an individual component.

```
# Give usermaint access to management and permission to execute all JMX Methods on
# UserManagement MBean and perform all actions for USER objects
ACL ALLOW usermaint ACCESS MANAGEMENT
ACL ALLOW usermaint ALL METHOD component="UserManagement"
ACL ALLOW usermaint ALL USER
ACL DENY ALL ALL METHOD component="UserManagement"
ACL DENY ALL ALL USER
...
... rules for other users
...
ACL DENY-LOG ALL ALL
```

8.3.3.3. Worked example 3 - Request/Response messaging

Suppose you wish to permission a system using a request/response paradigm. Two users: 'client' publishes requests; 'server' consumes the requests and generates a response. This example illustrates the permissioning of AMQP exchanges and queues.

```
# Allow client and server to connect to the virtual host.
ACL ALLOW client ACCESS VIRTUALHOST
ACL ALLOW server ACCESS VIRTUALHOST

# Client side
# Allow the 'client' user to publish requests to the request queue. As is the norm
# is required to create a temporary queue on which the server will respond.  Conse
# of the temporary queues and consumption of messages from it.
ACL ALLOW client CREATE QUEUE temporary="true"
ACL ALLOW client CONSUME QUEUE temporary="true"
ACL ALLOW client DELETE QUEUE temporary="true"
ACL ALLOW client BIND EXCHANGE name="amq.direct" temporary="true"
ACL ALLOW client UNBIND EXCHANGE name="amq.direct" temporary="true"
ACL ALLOW client PUBLISH EXCHANGE name="amq.direct" routingKey="example.RequestQue
```

```
# Server side
# Allow the 'server' user to consume from the request queue and publish a response
# client.  We also allow the server to create the request queue.
ACL ALLOW server CREATE QUEUE name="example.RequestQueue"
ACL ALLOW server CONSUME QUEUE name="example.RequestQueue"
ACL ALLOW server BIND EXCHANGE
ACL ALLOW server PUBLISH EXCHANGE name="amq.direct" routingKey="TempQueue*"

ACL DENY-LOG all all
```

8.3.3.4. Worked example 4 - firewall-like access control

This example illustrates how to set up an ACL that restricts the IP addresses and hostnames of messaging clients that can access a virtual host.

```
################
# Hostname rules
################

# Allow messaging clients from company1.com and company1.co.uk to connect
ACL ALLOW all ACCESS VIRTUALHOST from_hostname=".*\.company1\.com,.*\.company1\.co

# Deny messaging clients from hosts within the dev subdomain
ACL DENY-LOG all ACCESS VIRTUALHOST from_hostname=".*\.dev\.company1\.com"

##################
# IP address rules
##################

# Deny access to all users in the IP ranges 192.168.1.0-192.168.1.255 and 192.168.
# using the notation specified in RFC 4632, "Classless Inter-domain Routing (CIDR)
ACL DENY-LOG messaging-users ACCESS VIRTUALHOST \
  from_network="192.168.1.0/24,192.168.2.0/24"

# Deny access to all users in the IP ranges 192.169.1.0-192.169.1.255 and 192.169.
# using wildcard notation.
ACL DENY-LOG messaging-users ACCESS VIRTUALHOST \
  from_network="192.169.1.*,192.169.2.*"

ACL DENY-LOG all all
```

8.3.3.5. Worked example 5 - REST management ACL example

This example illustrates how to set up an ACL that restricts usage of REST management interfaces.

```
# allow to the users from webadmins group to change broker model
# this rule allows adding/removing/editing of Broker level objects:
# Broker, Group Provider, Authentication Provider, Port, Access Control Provider e
ACL ALLOW-LOG webadmins CONFIGURE BROKER
```

```
# allow to the users from webadmins group to perform
# create/update/delete on virtualhost node and children
ACL ALLOW-LOG webadmins CREATE VIRTUALHOSTNODE
ACL ALLOW-LOG webadmins UPDATE VIRTUALHOSTNODE
ACL ALLOW-LOG webadmins DELETE VIRTUALHOSTNODE
ACL ALLOW-LOG webadmins CREATE VIRTUALHOST
ACL ALLOW-LOG webadmins UPDATE VIRTUALHOST
ACL ALLOW-LOG webadmins DELETE VIRTUALHOST
ACL ALLOW-LOG webadmins CREATE QUEUE
ACL ALLOW-LOG webadmins UPDATE QUEUE
ACL ALLOW-LOG webadmins DELETE QUEUE
ACL ALLOW-LOG webadmins PURGE  QUEUE
ACL ALLOW-LOG webadmins CREATE EXCHANGE
ACL ALLOW-LOG webadmins DELETE EXCHANGE
ACL ALLOW-LOG webadmins BIND   EXCHANGE
ACL ALLOW-LOG webadmins UNBIND EXCHANGE

# allow to the users from webadmins group to create/update/delete groups on Group
ACL ALLOW-LOG webadmins CREATE GROUP
ACL ALLOW-LOG webadmins DELETE GROUP
ACL ALLOW-LOG webadmins UPDATE GROUP

# allow to the users from webadmins group to create/update/delete users for Authen
ACL ALLOW-LOG webadmins CREATE USER
ACL ALLOW-LOG webadmins DELETE USER
ACL ALLOW-LOG webadmins UPDATE USER

# allow to the users from webadmins group to move, copy, delete messagaes, and cle
# using REST management interfaces
ACL ALLOW-LOG webadmins UPDATE METHOD

# at the moment only the following UPDATE METHOD rules are supported by web manage
#ACL ALLOW-LOG webadmins UPDATE METHOD component="VirtualHost.Queue" name="moveMes
#ACL ALLOW-LOG webadmins UPDATE METHOD component="VirtualHost.Queue" name="copyMes
#ACL ALLOW-LOG webadmins UPDATE METHOD component="VirtualHost.Queue" name="deleteM
#ACL ALLOW-LOG webadmins UPDATE METHOD component="VirtualHost.Queue" name="clearQu

ACL DENY-LOG all all
```

8.4. Configuration Encryption

The Broker is capable of encrypting passwords and other security items stored in the Broker's configuration. This is means that items such as keystore/truststore passwords, JDBC passwords, and LDAP passwords can be stored in the configure in a form that is difficult to read.

The Broker ships with an encryptor implementation called `AESKeyFile`. This uses a securely generated random key of 256bit[1] to encrypt the secrets stored within a key file. Of course, the key itself must be guarded carefully, otherwise the passwords encrypted with it may be compromised. For this reason, the Broker ensures that the file's permissions allow the file to be read exclusively by the user account used for running the Broker.

[1]Java Cryptography Extension (JCE) Unlimited Strength required

Important

If the keyfile is lost or corrupted, the secrets will be irrecoverable.

8.4.1. Configuration

The `AESKeyFile` encyptor provider is enabled/disabled via the Broker attributes within the Web Management Console. On enabling the provider, any existing passwords within the configuration will be automatically rewritten in the encrypted form.

Note that passwords stored by the Authentication Providers PlainPasswordFile and. PlainPasswordFile with the external password files are *not* encrypted by the key. Use the Scram Authentication Managers instead; these make use of the Configuration Encryption when storing the users' passwords.

8.4.2. Alternate Implementations

If the `AESKeyFile` encryptor implementation does not meet the needs of the user, perhaps owing to the security standards of their institution, the `ConfigurationSecretEncrypter` interface is designed as an extension point. Users may implement their own implementation of ConfigurationSecretEncrypter perhaps to employ stronger encryption or delegating the storage of the key to an Enterprise Password Safe.

Chapter 9. Runtime

9.1. Log Files

The Broker uses the Apache Log4J [http://logging.apache.org/log4j/1.2/] Logging Framework for all logging activity.

In the Broker's shipped configuration, all logging is directed to log file `${QPID_WORK}/log/qpid.log`. The log file is not rotated and will be overwritten when the Broker restarts. Logging levels are configured in such a way that the log will comprise of:

- Opertional Log Events. These report key events in the lifecycle of objects (Broker start-up, Queue creation, Queue deletion etc) within the Broker. See Appendix C, *Operational Logging* for details of the formation of these messages.
- Queue Alert Events. These report when the queue thresholds have been breached. See Appendix D, *Queue Alerts* for details.
- Any Error and Warning conditions.

Logging can be reconfigured either by changing the logging configuration file `${QPID_HOME}/etc/log4j.xml` or at runtime using the Logging Management MBean, see Section 6.4.5, "The MBeans" for details.

9.1.1. Enabling Debug

It can be helpful to enable debug within the Broker in order to understand a problem more clearly. If this is required, debug can be enabled at runtime (without restarting the Broker) using the Logging Management MBean. The change can also be made by changing the log configuration file and restarting the Broker. Whichever mechanism is chosen, change the appender associated with `org.apache.qpid` from `WARN` to `DEBUG`.

Example 9.1. Changing the log4j.xml configuration file to enable debug

```
...
<logger additivity="true" name="org.apache.qpid">
    <level value="debug"/> <!-- change the level value from warn to debug -->
</logger>
...
```

Important

Running a production system at `DEBUG` level can have performance implications by slowing the Broker down. It can also generate large log files. Take care to revert the logging level back to `WARN` after the analysis is performed.

9.2. Disk Space Management

9.2.1. Producer Flow Control

9.2.1.1. General Information

The Java Broker supports a flow control mechanism to which can be used to prevent either a single queue or an entire virtualhost exceeding configured limits. These two mechanisms are described next.

9.2.1.2. Server Configuration

9.2.1.2.1. Configuring a Queue to use flow control

Flow control is enabled on a producer when it sends a message to a Queue which is "overfull". The producer flow control will be rescinded when all Queues on which a producer is blocking become "underfull". A Queue is defined as overfull when the size (in bytes) of the messages on the queue exceeds the *capacity* of the Queue. A Queue becomes "underfull" when its size becomes less than the *resume capacity*.

The capacity and resume capacity can be specified when the queue is created. This can be done using the Flow Control Settings within the Queue creation dialogue.

9.2.1.2.1.1. Broker Log Messages

There are four Broker log messages that may occur if flow control through queue capacity limits is enabled. Firstly, when a capacity limited queue becomes overfull, a log message similar to the following is produced

```
MESSAGE [vh(/test)/qu(MyQueue)] [vh(/test)/qu(MyQueue)] QUE-1003 : Overfull : Size
```

Then for each channel which becomes blocked upon the overful queue a log message similar to the following is produced:

```
MESSAGE [con:2(guest@anonymous(713889609)/test)/ch:1] [con:2(guest@anonymous(71388
```

When enough messages have been consumed from the queue that it becomes underfull, then the following log is generated:

```
MESSAGE [vh(/test)/qu(MyQueue)] [vh(/test)/qu(MyQueue)] QUE-1004 : Underfull : Siz
```

And for every channel which becomes unblocked you will see a message similar to:

```
MESSAGE [con:2(guest@anonymous(713889609)/test)/ch:1] [con:2(guest@anonymous(71388
```

Obviously the details of connection, virtual host, queue, size, capacity, etc would depend on the configuration in use.

9.2.1.2.2. Disk quota-based flow control

Flow control can also be triggered when a configured disk quota is exceeded. This is supported by the BDB and Derby virtualhosts.

This functionality blocks all producers on reaching the disk overflow limit. When consumers consume the messages, causing disk space usage to falls below the underflow limit, the producers are unblocked and continue working as normal.

Two limits can be configured:

overfull limit - the maximum space on disk (in bytes).

underfull limit - when the space on disk drops below this limit, producers are allowed to resume publishing.

The overfull and underful limit can be specified when a new virtualhost is created or an exiting virtualhost is edited. This can be done using the Store Overflow and Store Underfull settings within the virtual host creation and edit dialogue. If editing an existing virtualhost, the virtualhost must be restarted for the new values to take effect.

The disk quota functionality is based on "best effort" principle. This means the broker cannot guarantee that the disk space limit will not be exceeded. If several concurrent transactions are started before the limit is reached, which collectively cause the limit to be exceeded, the broker may allow all of them to be committed.

9.2.1.2.2.1. Broker Log Messages for quota flow control

There are two broker log messages that may occur if flow control through disk quota limits is enabled. When the virtual host is blocked due to exceeding of the disk quota limit the following message appears in the broker log

```
[vh(/test)/ms(BDBMessageStore)] MST-1008 : Store overfull, flow control will be en
```

When virtual host is unblocked after cleaning the disk space the following message appears in the broker log

```
[vh(/test)/ms(BDBMessageStore)] MST-1009 : Store overfull condition cleared
```

9.2.1.3. Client impact and configuration

If a producer sends to a queue which is overfull, the broker will respond by instructing the client not to send any more messages. The impact of this is that any future attempts to send will block until the broker rescinds the flow control order.

While blocking the client will periodically log the fact that it is blocked waiting on flow control.

```
WARN   Message send delayed by 5s due to broker enforced flow control
WARN   Message send delayed by 10s due to broker enforced flow control
```

After a set period the send will timeout and throw a JMSException to the calling code.

If such a JMSException is thrown, the message will not be sent to the broker, however the underlying Session may still be active - in particular if the Session is transactional then the current transaction will not be automatically rolled back. Users may choose to either attempt to resend the message, or to roll back any transactional work and close the Session.

Both the timeout delay and the periodicity of the warning messages can be set using Java system properties.

The amount of time (in milliseconds) to wait before timing out is controlled by the property qpid.flow_control_wait_failure.

The frequency at which the log message informing that the producer is flow controlled is sent is controlled by the system property qpid.flow_control_wait_notify_period.

Adding the following to the command line to start the client would result in a timeout of one minute, with warning messages every ten seconds:

```
-Dqpid.flow_control_wait_failure=60000
-Dqpid.flow_control_wait_notify_period=10000
```

9.3. Producer Transaction Timeout

9.3.1. General Information

The transaction timeout mechanism is used to control broker resources when clients producing messages using transactional sessions hang or otherwise become unresponsive, or simply begin a transaction and keep using it without ever calling Session#commit() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#commit].

Users can choose to configure an idleWarn or openWarn threshold, after which the identified transaction should be logged as a WARN level alert as well as (more importantly) an idleClose or openClose threshold after which the transaction and the connection it applies to will be closed.

This feature is particularly useful in environments where the owner of the broker does not have full control over the implementation of clients, such as in a shared services deployment.

The following section provide more details on this feature and its use.

9.3.2. Purpose

This feature has been introduced to address the scenario where an open transaction on the broker holds an open transaction on the persistent store. This can have undesirable consequences if the store does not time out or close long-running transactions, such as with BDB. This can can result in a rapid increase in disk usage size, bounded only by available space, due to growth of the transaction log.

9.3.3. Scope

Note that only MessageProducer [http://docs.oracle.com/javaee/6/api/javax/jms/MessageProducer.html] clients will be affected by a transaction timeout, since store transaction lifespan on a consumer only spans the execution of the call to Session#commit() and there is no scope for a long-lived transaction to arise.

It is also important to note that the transaction timeout mechanism is purely a JMS transaction timeout, and unrelated to any other timeouts in the Qpid client library and will have no impact on any RDBMS your application may utilise.

9.3.4. Effect

Full details of configuration options are provided in the sections that follow. This section gives a brief overview of what the Transaction Timeout feature can do.

9.3.4.1. Broker Logging and Connection Close

When the openWarn or idleWarn specified threshold is exceeded, the broker will log a WARN level alert with details of the connection and channel on which the threshold has been exceeded, along with the age of the transaction.

When the openClose or idleClose specified threshold value is exceeded, the broker will throw an exception back to the client connection via the ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html], log the action and then close the connection.

The example broker log output shown below is where the idleWarn threshold specified is lower than the idleClose threshold and the broker therefore logs the idle transaction 3 times before the close threshold is triggered and the connection closed out.

```
CHN-1008 : Idle Transaction : 13,116 ms
CHN-1008 : Idle Transaction : 14,116 ms
CHN-1008 : Idle Transaction : 15,118 ms
CHN-1003 : Close
```

The second example broker log output shown below illustrates the same mechanism operating on an open transaction.

```
CHN-1007 : Open Transaction : 12,406 ms
CHN-1007 : Open Transaction : 13,406 ms
CHN-1007 : Open Transaction : 14,406 ms
CHN-1003 : Close
```

9.3.4.2. Client Side Effect

After a Close threshold has been exceeded, the trigger client will receive this exception on its exception listener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html], prior to being disconnected: `org.apache.qpid.AMQConnectionClosedException: Error: Idle transaction timed out [error code 506: resource error]`

Any later attempt to use the connection will result in this exception being thrown:

```
Producer: Caught an Exception: javax.jms.IllegalStateException: Object org.apache.
    javax.jms.IllegalStateException: Object org.apache.qpid.client.AMQSession_0_8@
    at org.apache.qpid.client.Closeable.checkNotClosed(Closeable.java:70)
    at org.apache.qpid.client.AMQSession.checkNotClosed(AMQSession.java:555)
    at org.apache.qpid.client.AMQSession.createBytesMessage(AMQSession.java:573)
```

Thus clients must be able to handle this case successfully, reconnecting where required and registering an exception listener on all connections. This is critical, and must be communicated to client applications by any broker owner switching on transaction timeouts.

9.3.5. Configuration

9.3.5.1. Configuration

The transaction timeouts can be specified when a new virtualhost is created or an exiting virtualhost is edited.

We would recommend that only warnings are configured at first, which should allow broker administrators to obtain an idea of the distribution of transaction lengths on their systems, and configure production settings appropriately for both warning and closure. Ideally establishing thresholds should be achieved in a representative UAT environment, with clients and broker running, prior to any production deployment.

It is impossible to give suggested values, due to the large variation in usage depending on the applications using a broker. However, clearly transactions should not span the expected lifetime of any client application as this would indicate a hung client.

When configuring warning and closure timeouts, it should be noted that these only apply to message producers that are connected to the broker, but that a timeout will cause the connection to be closed - this disconnecting all producers and consumers created on that connection.

This should not be an issue for environments using Mule or Spring, where connection factories can be configured appropriately to manage a single MessageProducer object per JMS Session and Connection. Clients that use the JMS API directly should be aware that sessions managing both consumers and producers, or multiple producers, will be affected by a single producer hanging or leaving a transaction idle or open, and closed, and must take appropriate action to handle that scenario.

9.4. Handing Undeliverable Messages

9.4.1. Introduction

Messages that cannot be delivered successfully to a consumer (for instance, because the client is using a transacted session and rolls-back the transaction) can be made available on the queue again and then subsequently be redelivered, depending on the precise session acknowledgement mode and messaging model used by the application. This is normally desirable behaviour that contributes to the ability of a system to withstand unexpected errors. However, it leaves open the possibility for a message to be repeatedly redelivered (potentially indefinitely), consuming system resources and preventing the delivery of other messages. Such undeliverable messages are sometimes known as poison messages.

For an example, consider a stock ticker application that has been designed to consume prices contained within JMS TextMessages. What if inadvertently a BytesMessage is placed onto the queue? As the ticker application does not expect the BytesMessage, its processing might fail and cause it to roll-back the transaction, however the default behavior of the Broker would mean that the BytesMessage would be delivered over and over again, preventing the delivery of other legitimate messages, until an operator intervenes and removes the erroneous message from the queue.

Qpid has maximum delivery count and dead-letter queue (DLQ) features which can be used in concert to construct a system that automatically handles such a condition. These features are described in the following sections.

9.4.2. Maximum Delivery Count

Maximum delivery count is a property of a queue. If a consumer application is unable to process a message more than the specified number of times, then the broker will either route the message to a dead-letter queue (if one has been defined), or will discard the message.

In order for a maximum delivery count to be enforced, the consuming client *must* call Session#rollback() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#rollback()] (or Session#recover() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#recover()] if the session is not transacted). It is during the Broker's processing of Session#rollback() (or Session#recover()) that if a message has been seen at least the maximum number of times then it will move the message to the DLQ or discard the message.

If the consuming client fails in another manner, for instance, closes the connection, the message will not be re-routed and consumer application will see the same poison message again once it reconnects.

If the consuming application is using AMQP 0-9-1, 0-9, or 0-8 protocols, it is necessary to set the client system property `qpid.reject.behaviour` or connection or binding URL option `rejectbehaviour` to the value `server`.

It is possible to determine the number of times a message has been sent to a consumer via the Management interfaces, but is not possible to determine this information from a message client. Specifically, the optional JMS message header JMSXDeliveryCount is not supported.

Maximum Delivery Count can be specified when a new queue is created or using the the queue declare property x-qpid-maximum-delivery-count

9.4.3. Dead Letter Queues (DLQ)

A Dead Letter Queue (DLQ) acts as an destination for messages that have somehow exceeded the normal bounds of processing and is utilised to prevent disruption to flow of other messages. When a DLQ is enabled for a given queue if a consuming client indicates it no longer wishes the receive the message (typically by exceeding a Maximum Delivery Count) then the message is moved onto the DLQ and removed from the original queue.

The DLQ feature causes generation of a Dead Letter Exchange and a Dead Letter Queue. These are named convention QueueName*_DLE* and QueueName*_DLQ*.

DLQs can be enabled when a new queue is created or using the queue declare property x-qpid-dlq-enabled.

Avoid excessive queue depth

Applications making use of DLQs *should* make provision for the frequent examination of messages arriving on DLQs so that both corrective actions can be taken to resolve the underlying cause and organise for their timely removal from the DLQ. Messages on DLQs consume system resources in the same manner as messages on normal queues so excessive queue depths should not be permitted to develop.

9.5. Closing client connections on unroutable mandatory messages

9.5.1. Summary

Due to asynchronous nature of AMQP 0-8/0-9/0-9-1 protocols sending a message with a routing key for which no queue binding exist results in either message being bounced back (if it is mandatory or immediate) or discarded on broker side otherwise.

When a 'mandatory' message is returned back, the Qpid JMS client conveys this by delivering an *AMQNoRouteException* through the configured ExceptionListener on the Connection. This does not cause channel or connection closure, however it requires a special exception handling on client side in order to deal with *AMQNoRouteExceptions*. This could potentially be a problem when using various messaging frameworks (e.g. Mule) as they usually close the connection on receiving any JMSException.

In order to simplify application handling of scenarios where 'mandatory' messages are being sent to queues which do not actually exist, the Java Broker can be configured such that it will respond to this situation by closing the connection rather than returning the unroutable message to the client as it normally should. From the application perspective, this will result in failure of synchronous operations in progress such as a session commit() call.

Note

This feature affects only transacted sessions.

Qpid JMS client sends 'mandatory' messages when using Queue destinations and 'non-mandatory' messages when using Topic destinations.

9.5.2. Configuring *closeWhenNoRoute*

The Broker attribute *closeWhenNoRoute* can be set to specify this feature on broker side. By default, it is turned on. Setting *closeWhenNoRoute* to *false* switches it off.

Setting the *closeWhenNoRoute* in the JMS client connection URL can override the broker configuration on a connection specific basis, for example :

Example 9.2. Disable feature to close connection on unroutable messages with client URL

```
amqp://guest:guest@clientid/?brokerlist='tcp://localhost:5672'&closeWhenNoRoute='f
```

If no value is specified on the client the broker setting will be used. If client setting is specified then it will take precedence over the broker-wide configuration. If the client specifies and broker does not support this feature the warning will be logged.

9.6. Flow to Disk

Flow to disk limits the amount of heap memory that can be occupied by messages. Once this limit is reached any new transient messages and all existing transient messages will be transferred to disk. Newly arriving transient messages will continue to go to the disk until the cumulative size of all messages falls below the limit once again.

By default the Broker makes 40% of the max available memory for messages. This memory is divided between all the queues across all virtual hosts defined on the Broker with a percentage calculated according to their current queue size. These calculations are refreshed periodically by the housekeeping cycle.

For example if there are two queues, one containing 75MB and the second 100MB messages respectively and the Broker has 1GB heap memory with the default of 40% available for messages. The first queue will have a target size of 170MB and the second 230MB. Once 400MB is taken by messages, messages will begin to flow to disk. New messages will cease to flow to disk when their cumulative size falls beneath 400MB.

Flow to disk is configured by Broker context variable `broker.flowToDiskThreshold`. It is expressed as a size in bytes and defaults to 40% of the JVM maximum heap size.

Log message BRK-1014 is written when the feature activates. Once the total space of all messages decreases below the threshold, the message BRK-1015 is written to show that the feature is no longer active.

9.7. Background Recovery

On startup of the Broker, or restart of a Virtualhost, the Broker restores all durable queues and their messages from disk. In the Broker's default mode the Virtualhosts do not become active until this recovery process completes. If queues have a large number of entries, this may take considerable time. During this time no messaging can be performed.

The Broker has a background recovery feature allows the system to return to operation sooner. If enabled the recovery process takes place in the background allow producers and consumers to begin work earlier.

The feature respects the message delivery order requirements of standard queues, that is any messages arriving whilst the background recovery is in flight won't overtake older messages still to be recovered from disk. There is an exception for the out of order queue types whilst background recovery is in flight. For instance, with priority queues older lower priority messages may be delivered before newer, higher priority.

To activate the feature, set a context variable `use_async_message_store_recovery` at the desired Virtualhost, or at Broker or higher to enable the feature broker-wide.

Note

The background recovery feature does not write operational log messages to indicate its progress. This means messages MST-1004 and MST-1005 will not be seen.

9.8. Message Compression

The Java Broker supports[1] message compression. This feature works in co-operation with Qpid Clients implementing the same feature.

Once the feature is enabled (using Broker context variable *broker.messageCompressionEnabled*), the Broker will advertise support for the message compression feature to the client at connection time. This allows clients to opt to turn on message compression, allowing message payload sizes to be reduced.

If the Broker has connections from clients who have message compression enabled and others who do not, it will internally, on-the-fly, decompress compressed messages when sending to clients without support and conversely, compress uncomressed messages when sending to clients who do.

The Broker has a threshold below which it will not consider compressing a message, this is controlled by Broker content variable (`connection.messageCompressionThresholdSize`) and expresses a size in bytes.

This feature *may* have a beneficial effect on performance by:

- Reducing the number of bytes transmitted over the wire, both between Client and Broker, and in the HA case, Broker to Broker, for replication purposes.
- Reducing storage space when data is at rest within the Broker, both on disk and in memory.

[1]Message compression is not yet supported for the 1.0 protocol.

Of course, compression and decompression is computationally expensive. Turning on the feature may have a negative impact on CPU utilization on Broker and/or Client. Also for small messages payloads, message compression may increase the message size. It is recommended to test the feature with representative data.

9.9. Connection Limits

Each connection to the Broker consumes resources while it is connected. In order to protect the Broker against malfunctioning (or malicious) client processes, it is possible to limit the number of connections that can be active on any given port.

Connection limits on AMQP ports are controlled by an attribute "maxOpenConnections" on the port. By default this takes the value of the context variable `qpid.port.max_open_connections` which in itself is defaulted to the value `-1` meaning there is no limit.

If the interpolated value of `maxOpenConnections` on an AMQP port is a positive integer, then when that many active connections have been established no new connections will be allowed (until an existing connection has been closed). Any such rejection of a connection will be accompanied by the operational log message PRT-1005.

The context variable `qpid.port.open_connections_warn_percent` can be used to control when a warning log message is generated as the number of open connections approaches the limit for the port. The default value of this variable is `80` meaning that if more the number of open connections to the port has exceeded 80% of the given limit then the operatinal log message PRT-1004 will be generated.

Chapter 10. High Availability

10.1. General Introduction

The term High Availability (HA) usually refers to having a number of instances of a service such as a Message Broker available so that should a service unexpectedly fail, or requires to be shutdown for maintenance, users may quickly connect to another instance and continue their work with minimal interruption. HA is one way to make a overall system more resilient by eliminating a single point of failure from a system.

HA offerings are usually categorised as **Active/Active** or **Active/Passive**. An Active/Active system is one where all nodes within the group are usually available for use by clients all of the time. In an Active/Passive system, one only node within the group is available for use by clients at any one time, whilst the others are in some kind of standby state, awaiting to quickly step-in in the event the active node becomes unavailable.

10.2. Overview of HA within the Java Broker

The Java Broker provides a HA implementation offering an **Active/Passive** mode of operation. When using HA, many instances of the Java Broker work together to form an high availability group of two or more nodes.

The remainder of this section now talks about the specifics of how HA is achieved in terms of the concepts introduced earlier in this book.

The Virtualhost is the unit of replication. This means that any *durable* queues, exchanges, and bindings belonging to that virtualhost, any *persistent* messages contained within the queues and any attribute settings applied to the virtualhost itself are automatically replicated to all nodes within the group.[1]

It is the Virtualhost Nodes (from different Broker instances) that join together to form a group. The virtualhost nodes collectively to coordinate the group: they organise replication between the master and replicas and conduct elections to determine who becomes the new master in the event of the old failing.

When a virtualhost node is in the *master* role, the virtualhost beneath it is available for messaging work. Any write operations sent to the virtualhost are automatically replicated to all other nodes in group.

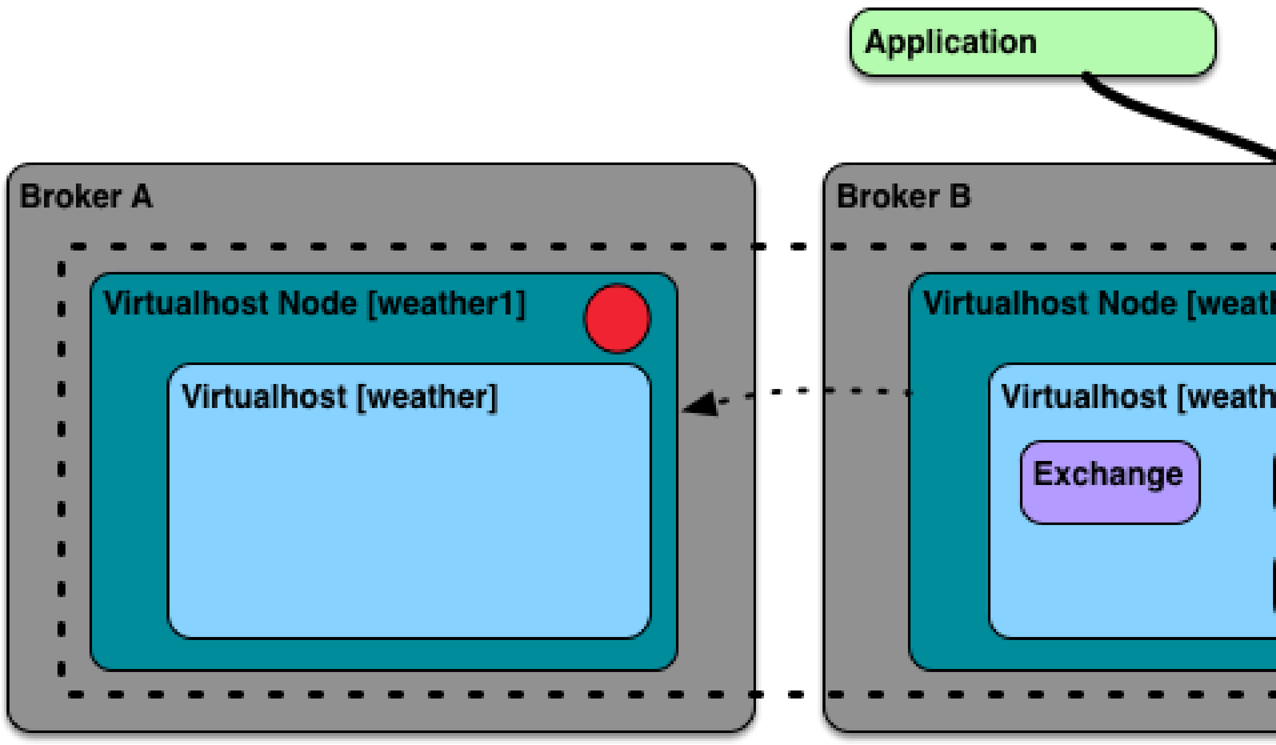
When a virtualhost node is in the *replica* role, the virtualhost beneath it is always unavailable for message work. Any attempted connections to a virtualhost in this state are automatically turned away, allowing a messaging client to discover where the master currently resides. When in replica role, the node sole responsibility is to consume a replication stream in order that it remains up to date with the master.

Messaging clients discover the active virtualhost.This can be achieved using a static technique (for instance, a failover url (a feature of a Qpid Java Client)), or a dynamic one utilising some kind of proxy or virtual IP (VIP).

The figure that follows illustrates a group formed of three virtualhost nodes from three separate Broker instances. A client is connected to the virtualhost node that is in the master role. The two virtualhost nodes `weather1` and `weather3` are replicas and are receiving a stream of updates.

[1]Transient messages and messages on non-durable queues are not replicated.

Figure 10.1. 3-node group deployed across three Brokers.



Currently, the only virtualhost/virtualhost node type offering HA is BDB HA. Internally, this leverages the HA capabilities of the Berkeley DB JE edition. BDB JE is an optional dependency of the Broker.

Note

The Java Broker HA solution is incompatible with the HA solution offered by the CPP Broker. It is not possible to co-locate Java and CPP Brokers within the same group.

10.3. Creating a group

This section describes how to create a group. At a high level, creating a group involves first creating the first node standalone, then creating subsequent nodes referencing the first node so the nodes can introduce themselves and gradually the group is built up.

A group is created through either Web Management or the REST API. These instructions presume you are using Web Management. To illustrate the example it builds the group illustrated in figure Figure 10.1, “3-node group deployed across three Brokers.”

1. Install a Broker on each machine that will be used to host the group. As messaging clients will need to be able to connect to and authentication to all Brokers, it usually makes sense to choose a common authentication mechanism e.g. Simple LDAP Authentication, External with SSL client authentication or Kerberos.

2. Select one Broker instance to host the first node instance. This choice is an arbitrary one. The node is special only whilst creating group. Once creation is complete, all nodes will be considered equal.

3. Click the `Add` button on the Virtualhost Panel on the Broker tab.

   a. Give the Virtualhost node a unique name e.g. `weather1`. The name must be unique within the group and unique to that Broker. It is best if the node names are chosen from a different nomenclature than the machine names themselves.

   b. Choose `BDB_HA` and select `New group`

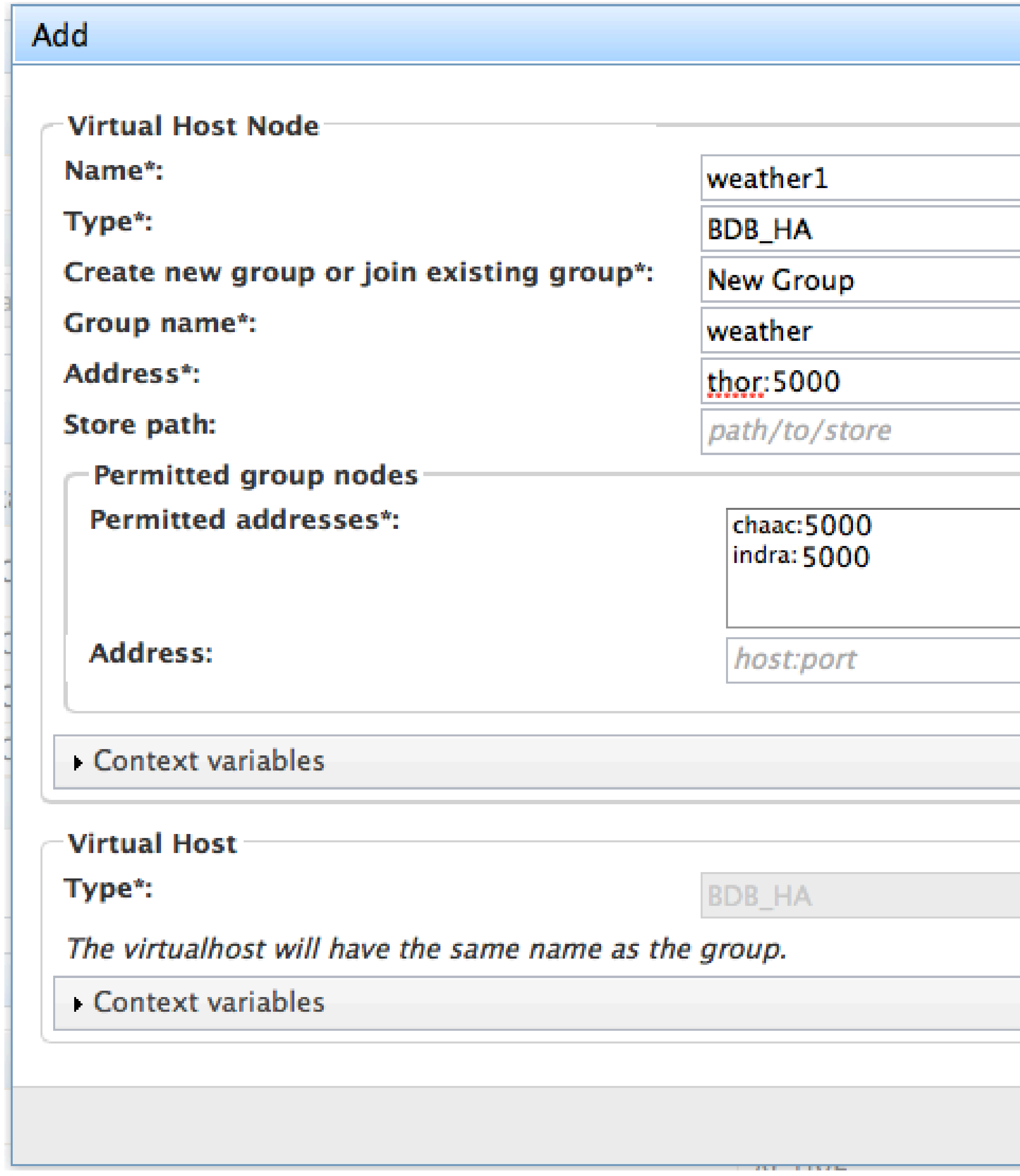
   c. Give the group a name e.g. `weather`. The group name must be unique and will be the name also given to the virtualhost, so this is the name the messaging clients will use in their connection url.

   d. Give the address of this node. This is an address on this node's host that will be used for replication purposes. The hostname *must* be resolvable by all the other nodes in the group. This is separate from the address used by messaging clients to connect to the Broker. It is usually best to choose a symbolic name, rather than an IP address.

   e. Now add the node addresses of all the other nodes that will form the group. In our example we are building a three node group so we give the node addresses of `chaac:5000` and `indra:5000`.

   f. Click Add to create the node. The virtualhost node will be created with the virtualhost. As there is only one node at this stage, the role will be master.

Figure 10.2. Creating 1st node in a group


Add

Virtual Host Node

Name*: weather1

Type*: BDB_HA

Create new group or join existing group*: New Group

Group name*: weather

Address*: thor:5000

Store path: path/to/store

Permitted group nodes

Permitted addresses*: chaac:5000 indra:5000

Address: host:port

Context variables

Virtual Host

Type*: BDB_HA

The virtualhost will have the same name as the group.

Context variables

4. Now move to the second Broker to be the group. Click the `Add` button on the Virtualhost Panel on the Broker tab of the second Broker.

   a. Give the Virtualhost node a unique name e.g. `weather2`.

b. Choose `BDB_HA` and choose `Existing group`

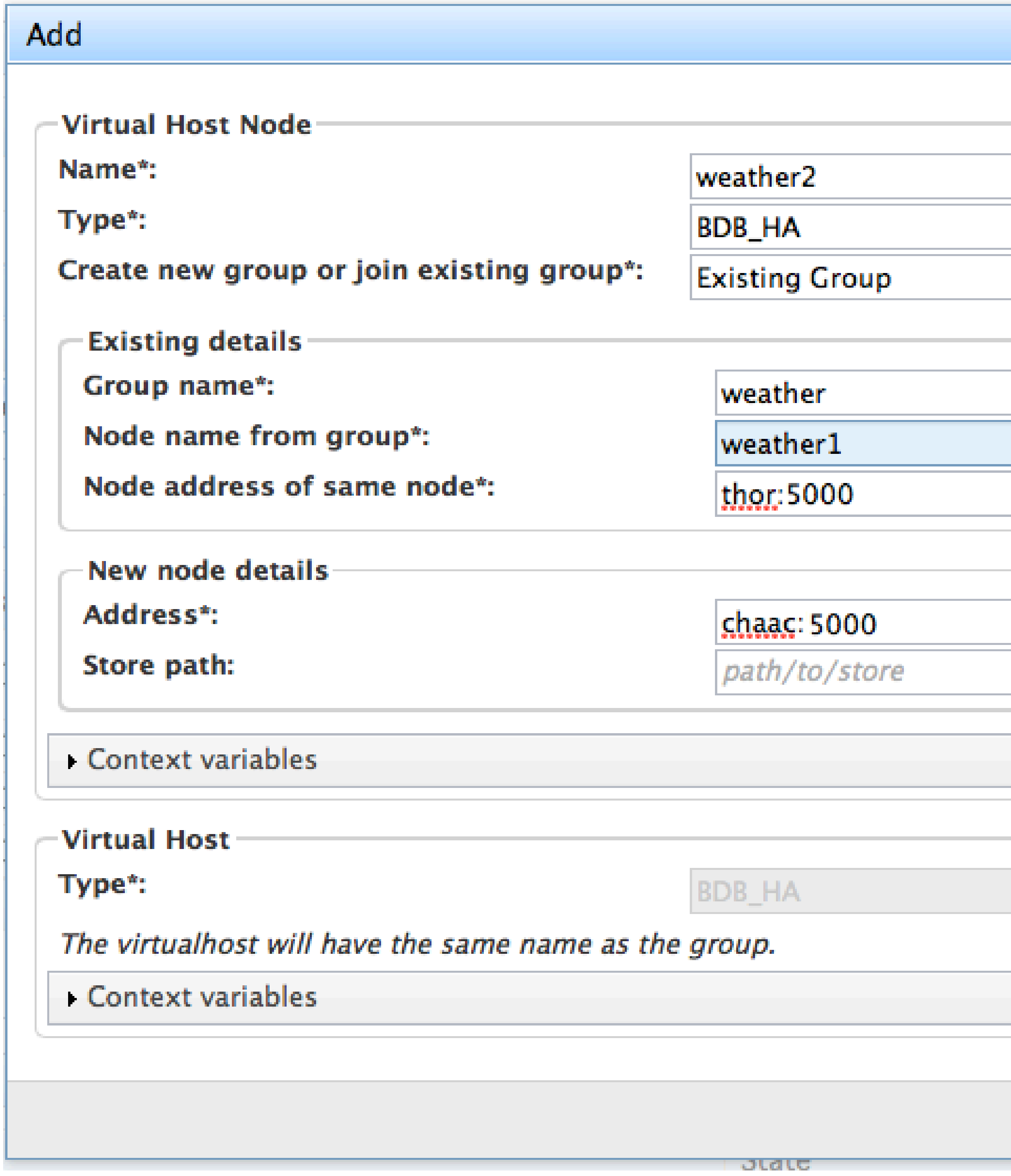
c. Give the details of the *existing node*. Following our example, specify `weather`, `weather1` and `thor:5000`

d. Give the address of this node.

e. Click Add to create the node. The node will use the existing details to contact it and introduce itself into the group. At this stage, the group will have two nodes, with the second node in the replica role.

f. Repeat these steps until you have added all the nodes to the group.

Figure 10.3. Adding subsequent nodes to the group


Add

Virtual Host Node

Name*:	weather2
Type*:	BDB_HA
Create new group or join existing group*:	Existing Group

Existing details

Group name*:	weather
Node name from group*:	weather1
Node address of same node*:	thor:5000

New node details

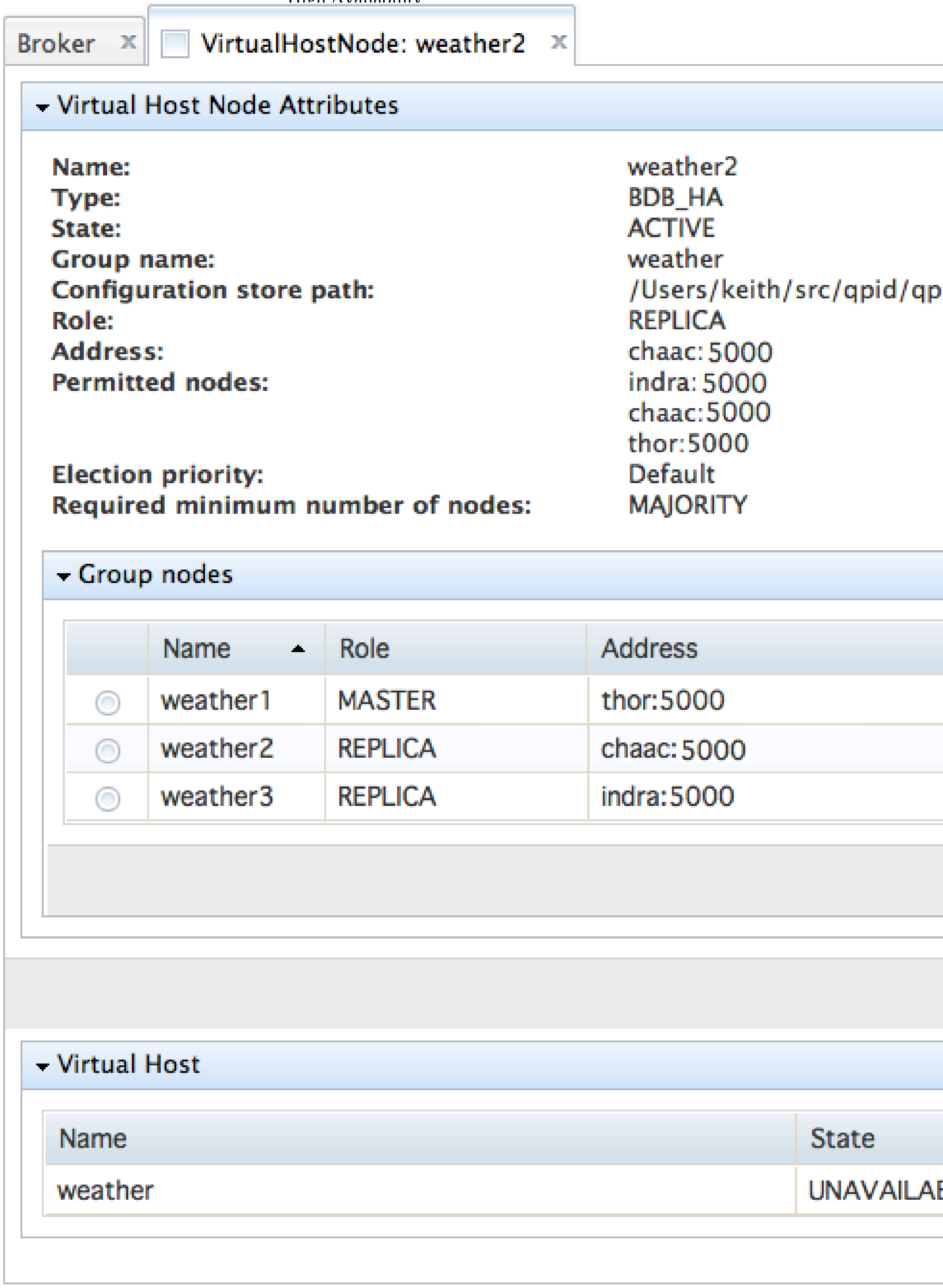
Address*:	chaac:5000
Store path:	path/to/store

Virtual Host

Type*:	BDB_HA

The virtualhost will have the same name as the group.

The group is now formed and is ready for us. Looking at the virtualhost node of any of the nodes shows a complete view of the whole group.



Name:	weather2
Type:	BDB_HA
State:	ACTIVE
Group name:	weather
Configuration store path:	/Users/keith/src/qpid/qp
Role:	REPLICA
Address:	chaac:5000
Permitted nodes:	indra:5000 chaac:5000 thor:5000
Election priority:	Default
Required minimum number of nodes:	MAJORITY

	Name	Role	Address
○	weather1	MASTER	thor:5000
○	weather2	REPLICA	chaac:5000
○	weather3	REPLICA	indra:5000

Name	State
weather	UNAVAILAE

10.4. Behaviour of the Group

This section first describes the behaviour of the group in its default configuration. It then goes on to talk about the various controls that are available to override it. It describes the controls available that affect the durability [http://en.wikipedia.org/wiki/ACID#Durability] of transactions and the data consistency between the master and replicas and thus make trade offs between performance and reliability.

10.4.1. Default Behaviour

Let's first look at the behaviour of a group in default configuration.

In the default configuration, for any messaging work to be done, there must be at least *quorum* nodes present. This means for example, in a three node group, this means there must be at least two nodes available.

When a messaging client sends a transaction, it can be assured that, before the control returns back to his application after the commit call that the following is true:

- At the master, the transaction is *written to disk and OS level caches are flushed* meaning the data is on the storage device.
- At least quorum minus 1 replicas, *acknowledge the receipt of transaction*. The replicas will write the data to the storage device sometime later.

If there were to be a master failure immediately after the transaction was committed, the transaction would be held by at least quorum minus one replicas. For example, if we had a group of three, then we would be assured that at least one replica held the transaction.

In the event of a master failure, if quorum nodes remain, those nodes hold an election. The nodes will elect master the node with the most recent transaction. If two or more nodes have the most recent transaction the group makes an arbitrary choice. If quorum number of nodes does not remain, the nodes cannot elect a new master and will wait until nodes rejoin. You will see later that manual controls are available allow service to be restored from fewer than quorum nodes and to influence which node gets elected in the event of a tie.

Whenever a group has fewer than quorum nodes present, the virtualhost will be unavailable and messaging connections will be refused. If quorum disappears at the very moment a messaging client sends a transaction that transaction will fail.

You will have noticed the difference in the synchronization policies applied the master and the replicas. The replicas send the acknowledgement back before the data is written to disk. The master synchronously writes the transaction to storage. This is an example of a trade off between durability and performance. We will see more about how to control this trade off later.

10.4.2. Synchronization Policy

The *synchronization policy* dictates what a node must do when it receives a transaction before it acknowledges that transaction to the rest of the group.

The following options are available:

- *SYNC*. The node must write the transaction to disk and flush any OS level buffers before sending the acknowledgement. SYNC is offers the highest durability but offers the least performance.
- *WRITE_NO_SYNC*. The node must write the transaction to disk before sending the acknowledgement. OS level buffers will be flush as some point later. This typically provides an assurance against failure of the application but not the operating system or hardware.

- *NO_SYNC*. The node immediately sends the acknowledgement. The transaction will be written and OS level buffers flushed as some point later. NO_SYNC offers the highest performance but the lowest durability level. This synchronization policy is sometimes known as *commit to the network*.

It is possible to assign a one policy to the master and a different policy to the replicas. These are configured as attributes on the virtualhost. By default the master uses *SYNC* and replicas use *NO_SYNC*.

10.4.3. Node Priority

Node priority can be used to influence the behaviour of the election algorithm. It is useful in the case were you want to favour some nodes over others. For instance, if you wish to favour nodes located in a particular data centre over those in a remote site.

The following options are available:

- *Highest*. Nodes with this priority will be more favoured. In the event of two or more nodes having the most recent transaction, the node with this priority will be elected master. If two or more nodes have this priority the algorithm will make an arbitrary choice.
- *High*. Nodes with this priority will be favoured but not as much so as those with Highest.
- *Normal*. This is default election priority.
- *Never*. The node will never be elected *even if the node has the most recent transaction*. The node will still keep up to date with the replication stream and will still vote itself, but can just never be elected.

Node priority is configured as an attribute on the virtualhost node and can be changed at runtime and is effective immediately.

Important

Use of the Never priority can lead to transaction loss. For example, consider a group of three where replica-2 is marked as Never. If a transaction were to arrive and it be acknowledged only by Master and Replica-2, the transaction would succeed. Replica 1 is running behind for some reason (perhaps a full-GC). If a Master failure were to occur at that moment, the replicas would elect Replica-1 even though Replica-2 had the most recent transaction.

Transaction loss is reported by message HA-1014.

10.4.4. Required Minimum Number Of Nodes

This controls the required minimum number of nodes to complete a transaction and to elect a new master. By default, the required number of nodes is set to *Default* (which signifies quorum).

It is possible to reduce the required minimum number of nodes. The rationale for doing this is normally to temporarily restore service from fewer than quorum nodes following an extraordinary failure.

For example, consider a group of three. If one node were to fail, as quorum still remained, the system would continue work without any intervention. If the failing node were the master, a new master would be elected.

What if a further node were to fail? Quorum no longer remains, and the remaining node would just wait. It cannot elect itself master. What if we wanted to restore service from just this one node?

In this case, Required Number of Nodes can be reduced to 1 on the remain node, allowing the node to elect itself and service to be restored from the singleton. Required minimum number of nodes is configured as an attribute on the virtualhost node and can be changed at runtime and is effective immediately.

Important

The attribute must be used cautiously. Careless use will lead to lost transactions and can lead to a split-brain [http://en.wikipedia.org/wiki/Split-brain_(computing)] in the event of a network partition. If used to temporarily restore service from fewer than quorum nodes, it is *imperative* to revert it to the Default value as the failed nodes are restored.

Transaction loss is reported by message HA-1014.

10.4.5. Designated Primary

This attribute applies to the groups of two only.

In a group of two, if a node were to fail then in default configuration work will cease as quorum no longer exists. A single node cannot elect itself master.

The designated primary flag allows a node in a two node group to elect itself master and to operate sole. Designated Primary is configured as an attribute on the virtualhost node and can be changed at runtime and is effective immediately.

For example, consider a group of two where the master fails. Service will be interrupted as the remaining node cannot elect itself master. To allow it to become master, apply the designated primary flag to it. It will elect itself master and work can continue, albeit from one node.

Important

It is imperative not to allow designated primary to be set on both nodes at once. To do so will mean, in the event of a network partition, a split-brain [http://en.wikipedia.org/wiki/Split-brain_(computing)] will occur.

Transaction loss is reported by message HA-1014.

10.5. Node Operations

10.5.1. Lifecycle

Virtualhost nodes can be stopped, started and deleted.

- *Stop*

  Stopping a master node will cause the node to temporarily leave the group. Any messaging clients will be disconnected and any in-flight transaction rollbacked. The remaining nodes will elect a new master if quorum number of nodes still remains.

  Stopping a replica node will cause the node to temporarily leave the group too. Providing quorum still exists, the current master will continue without interruption. If by leaving the group, quorum no longer exists, all the nodes will begin waiting, disconnecting any messaging clients, and the virtualhost will become unavailable.

  A stopped virtualhost node is still considered to be a member of the group.

- *Start*

  Starting a virtualhost node allows it to rejoin the group.

If the group already has a master, the node will catch up from the master and then become a replica once it has done so.

If the group did not have quorum and so had no master, but the rejoining of this node means quorum now exists, an election will take place. The node with the most up to date transaction will become master unless influenced by the priority rules described above.

Note

The length of time taken to catch up will depend on how long the node has been stopped. The worst case is where the node has been stopped for more than one hour. In this case, the master will perform an automated `network restore`. This involves streaming all the data held by the master over to the replica. This could take considerable time.

- *Delete*

  A virtualhost node can be deleted. Deleting a node permanently removes the node from the group. The data stored locally is removed but this does not affect the data held by the remainder of the group.

Note

The names of deleted virtualhost node cannot be reused within a group.

It is also possible to add nodes to an existing group using the procedure described above.

10.5.2. Transfer Master

This operation allows the mastership to be moved from node to node. This is useful for restoring a business as usual state after a failure.

When using this function, the following occurs.

1. The system first gives time for the chosen new master to become reasonable up to date.
2. It then suspends transactions on the old master and allows the chosen node to become up to date.
3. The suspended transactions are aborted and any messaging clients connected to the old master are disconnected.
4. The chosen master becomes the new master. The old master becomes a replica.
5. Messaging clients reconnect the new master.

10.6. Client failover

As mentioned above, the clients need to be able to find the location of the active virtualhost within the group.

Clients can do this using a static technique, for example , utilising the failover feature of the Qpid connection url [../../Programming-In-Apache-Qpid/html/QpidJNDI.html#section-jms-connection-url] where the client has a list of all the nodes, and tries each node in sequence until it discovers the node with the active virtualhost.

Another possibility is a dynamic technique utilising a proxy or Virtual IP (VIP). These require other software and/or hardware and are outside the scope of this document.

10.7. Qpid JMX API for HA

The Qpid JMX API for HA is now deprecated. New users are recommended to use the REST API.

10.8. Disk space requirements

In the case where node in a group are down, the master must keep the data they are missing for them to allow them to return to the replica role quickly.

By default, the master will retain up to 1hour of missed transactions. In a busy production system, the disk space occupied could be considerable.

This setting is controlled by virtualhost context variable `je.rep.repStreamTimeout`.

10.9. Network Requirements

The HA Cluster performance depends on the network bandwidth, its use by existing traffic, and quality of service.

In order to achieve the best performance it is recommended to use a separate network infrastructure for the Qpid HA Nodes which might include installation of dedicated network hardware on Broker hosts, assigning a higher priority to replication ports, installing a group in a separate network not impacted by any other traffic.

10.10. Security

The replication stream between the master and the replicas is insecure and can be intercepted by anyone having access to the replication network.

In order to reduce the security risks the entire HA group is recommended to run in a separate network protected from general access and/or utilise SSH-tunnels/IPsec.

10.11. Backups

It is recommend to use the hot backup script to periodically backup every node in the group. Section 11.2.2, “BDB-HA”.

10.12. Reset Group Information

BDB JE internally stores details of the group within its database. There are some circumstances when resetting this information is useful.

- Copying data between environments (e.g. production to UAT)
- Some disaster recovery situations where a group must be recreated on new hardware

This is not an normal operation and is not usually required

The following command replaces the group table contained within the JE logs files with the provided information.

Example 10.1. Resetting of replication group with `DbResetRepGroup`

`java` -cp je-5.0.104.jar com.sleepycat.je.rep.util.DbResetRepGroup -h *`path/to/jelogfiles`* -groupName *`newgroupname`* -nodeName *`newnodename`* -nodeHostPort *`newhostname:5000`*

The modified log files can then by copied into `${QPID_WORK}/<nodename>/config` directory of a target Broker. Then start the Broker, and add a BDB HA Virtualhost node specify the same group name, node name and node address. You will then have a group with a single node, ready to start re-adding additional nodes as described above.

Chapter 11. Backup And Recovery

11.1. Broker

To perform a complete backup whilst the Broker is shutdown, simply copy all the files the exist beneath `${QPID_WORK}`, assuming all virtualhost nodes and virtualhost are in their standard location, this will copy all configuration and persistent message data.

There is currently no safe mechanism to take a complete copy of the entire Broker whilst it is running.

11.2. Virtualhost Node

To perform a complete backup of a Virtualhost node whilst it is stopped (or Broker down), simply copy all the files the exist beneath `${QPID_WORK}/<nodename>/config`, assuming the virtualhost node is in the standard location. This will copy all configuration that belongs to that virtualhost node.

The technique for backing up a virtualhost node whilst it is running depends on its type.

11.2.1. BDB

Qpid Broker distribution includes the "hot" backup utility `backup.sh` which can be found at broker bin folder. This utility can perform the backup when broker is running.

`backup.sh` script invokes `org.apache.qpid.server.store.berkeleydb.BDBBackup` to do the job.

You can also run this class from command line like in an example below:

Example 11.1. Performing store backup by using `BDBBackup` class directly

`java` -cp qpid-bdbstore-0.32-SNAPSHOT.jar org.apache.qpid.server.store.berkeleydb.BDBBackup -fromdir *`${QPID_WORK}/<nodename>/config`* -todir *`path/to/backup/folder`*

In the example above BDBBackup utility is called from qpid-bdbstore-0.32-SNAPSHOT.jar to backup the store at `${QPID_WORK}/<nodename>/config` and copy store logs into `path/to/backup/folder`.

Linux and Unix users can take advantage of `backup.sh` bash script by running this script in a similar way.

Example 11.2. Performing store backup by using `backup.sh` bash script

`backup.sh` -fromdir *`${QPID_WORK}/<nodename>/config`* -todir *`path/to/backup/folder`*

11.2.2. BDB-HA

See Section 11.2.1, “BDB”

11.2.3. Derby

Not yet supported

11.2.4. JDBC

The responsibility for backup is delegated to the database server itself. See the documentation accompanying it. Any technique that takes a consistent snapshot of the database is acceptable.

11.2.5. JSON

JSON stores its config in a single text file. It can be safely backed up using standard command line tools.

11.3. Virtualhost

To perform a complete backup of a Virtualhost whilst it is stopped (or Broker down), simply copy all the files the exist beneath `${QPID_WORK}/<name>/messages`, assuming the virtualhost is in the standard location. This will copy all messages that belongs to that virtualhost.

The technique for backing up a virtualhost whilst it is running depends on its type.

11.3.1. BDB

Use the same backup utility described above, but use the path `${QPID_WORK}/<name>/messages` instead.

11.3.2. Derby

Not yet supported

11.3.3. JDBC

The responsibility for backup is delegated to the database server itself. See the documentation accompanying it. Any technique that takes a consistent snapshot of the database is acceptable.

11.3.4. Provided

The contents of the virtualhost will be backed up as part of virtualhost node that contains it.

11.3.5. BDB-HA

The contents of the virtualhost will be backed up as part of virtualhost node that contains it.

Appendix A. Environment Variables

The following table describes the environment variables understood by the Qpid scripts contained within the `/bin` directory within the Broker distribution.

To take effect, these variables must be set within the shell (and exported - if using Unix) before invoking the script.

Table A.1. Environment variables

Environment variable	Default	Purpose
QPID_HOME	None	The variable used to tell the Broker its installation directory. It must be an absolute path. This is used to determine the location of Qpid's dependency JARs and some configuration files. Typically the value of this variable will look similar to `c:\qpid\qpid-broker-0.32-SNAPSHOT` (Windows) or `/usr/local/qpid/qpid-broker-0.32-SNAPSHOT` (Unix). The installation prefix will differ from installation to installation. If not set, a value for `QPID_HOME` is derived from the location of the script itself.
QPID_WORK	User's home directory	Used as the default root directory for any data written by the Broker. This is the default location for any message data written to persistent stores and the Broker's log file. For example, `QPID_WORK=/var/qpidwork`.
QPID_OPTS	None	This is the preferred mechanism for passing Java system properties to the Broker. The value must be a list of system properties each separate by a space. `-D`*`name1`*`=`*`value1`* `-D`*`name2`*`=`*`value2`*.
QPID_JAVA_GC	`-XX:+HeapDumpOnOutOfMemoryError -XX:+UseConcMarkSweepGC`	This is the preferred mechanism for customising garbage collection behaviour. The value should contain valid garbage collection options(s) for the target JVM.

Environment variable	**Default**	**Purpose**
		Refer to the JVM's documentation for details.
QPID_JAVA_MEM	`-Xmx2g`	This is the preferred mechanism for customising the size of the JVM's heap memory. The value should contain valid memory option(s) for the target JVM. Oracle JVMs understand `-Xmx` to specify a maximum heap size and `-Xms` an initial size. For example, `QPID_JAVA_MEM=-Xmx6g` would set a maximum heap size of 6GB. Refer to the JVM's documentation for details.
JAVA_OPTS	None	This is the preferred mechanism for passing any other JVM options. This variable is commonly used to pass options for diagnostic purposes, for instance to turn on verbose GC. `-verbose:gc`. Refer to the JVM's documentation for details.

Appendix B. System Properties

The following table describes the Java system properties used by the Broker to control various optional behaviours.

The preferred method of enabling these system properties is using the `QPID_OPTS` environment variable described in the previous section.

Table B.1. System properties

System property	Default	Purpose
qpid.broker_heartbeat_timeout_factor	2	Factor to determine the maximum length of that may elapse between heartbeats being received from the peer before a connection is deemed to have been broken.
qpid.broker_dead_letter_exchange_suffix	_DLE	Used with the Section 9.4.3, “Dead Letter Queues (DLQ)” feature. Governs the suffix used when generating a name for a Dead Letter Exchange.
qpid.broker_dead_letter_queue_suffix	_DLQ	Used with the Section 9.4.3, “Dead Letter Queues (DLQ)” feature. Governs the suffix used when generating a name for a Dead Letter Queue.
qpid.broker_msg_auth	false	If set true, the Broker ensures that the user id of each received message matches the user id of the producing connection. If this check fails, the message is returned to the producer's connection with a 403 (Access Refused) error code. This is check is currently not enforced when using AMQP 0-10 and 1-0 protocols.
qpid.broker_status_updates	true	If set true, the Broker will produce operational logging messages.
qpid.broker_default_supported_protocol_version_reply	none	Used during protocol negotiation. If set, the Broker will offer this AMQP version to a client requesting an AMQP protocol that is not supported by the Broker. If not set, the Broker offers the highest protocol version it supports.
qpid.broker_disabled_features	none	Allows optional Broker features to be disabled. Currently understood feature names are: `qpid.jms-selector`

System property	Default	Purpose
		Feature names should be comma separated.
qpid.broker_frame_size	65536	Maximum AMQP frame size supported by the Broker.
qpid.broker_jmx_method_rights_infer_all_access	true	Used when using ACLs and the JMX management interface. If set true, the METHOD object permission is sufficient to allow the user to perform the operation. If set false, the user will require both the METHOD object permission and the underlying object permission too (for instance QUEUE object permission if performing management operations on a queue). If the user is not granted both permissions, the operation will be denied.
qpid.broker_jmx_use_custom_rmi_socket_factory	true	Applicable to the JMX management interface. If true, the Broker creates a custom RMI socket factory that is secured from changes outside of the JVM. If false, a standard RMI socket factory is used. It is not recommended to change this property from its default value.
qpid.broker_log_records_buffer_size	4096	Controls the number of recent Broker log entries that remain viewable online via the HTTP Management interface.

Appendix C. Operational Logging

The Broker will, by default, produce structured log messages in response to key events in the lives of objects within the Broker. These concise messages are designed to allow the user to understand the actions of the Broker in retrospect. This is valuable for problem diagnosis and provides a useful audit trail.

Each log message includes details of the entity causing the action (e.g. a management user or messaging client connection), the entity receiving the action (e.g. a queue or connection) and a description of operation itself.

The log messages have the following format:

```
[Actor] {[Subject]} [Message Id] [Message Text]
```

Where:

- `Actor` is the entity within the Broker that is *performing* the action. There are actors corresponding to the Broker itself, Management, Connection, and Channels. Their format is described in the table below.
- `Subject` (optional) is the entity within the Broker that is *receiving* the action. There are subjects corresponding to the Connections, Channels, Queues, Exchanges, Subscriptions, and Message Stores. Their format is described in the table below.

  Some actions are reflexive, in these cases the Actor and Subject will be equal.
- `Message Id` is an identifier for the type of message. It has the form three alphas and four digits separated by a hyphen `AAA-9999`.
- `Message Text` is a textual description

To illustrate, let's look at two examples.

`CON-1001` is used when a messages client makes an AMQP connection. The connection actor (`con`) provides us with details of the peer's connection: the user id used by the client (myapp1), their IP, ephemeral port number and the name of the virtual host. The message text itself gives us further details about the connection: the client id, the protocol version in used, and details of the client's qpid library.

```
[con:8(myapp1@/127.0.0.1:52851/default)] CON-1001 : Open : Client ID : clientid :
             Protocol Version : 0-10 : Client Version : 0.32-SNAPSHOT : Client Pro
```

`QUE-1001` is used when a queue is created. The connection actor `con` tells us details of the connection performing the queue creation: the user id used by the client (myapp1), the IP, ephemeral port number and the name of the virtual host. The queue subject tells use the queue's name (myqueue) and the virtualhost. The message itself tells us more information about the queue that is being created.

```
[con:8(myapp1@/127.0.0.1:52851/default)/ch:0] [vh(/default)/qu(myqueue)] QUE-1001
```

The first two tables that follow describe the actor and subject entities, then the later provide a complete catalogue of all supported messages.

Table C.1. Actors Entities

Actor Type	Format and Purpose
Broker	[Broker]
	Used during startup and shutdown

Actor Type	Format and Purpose
Management	[mng:*userid*(*clientip*:*ephemeralport*)]
	Used for operations performed by the either the JMX or Web Management interfaces.
Connection	[con:*connectionnumber*(*userid*@/ *clientip*:*ephemeralport*/*virtualhostname*)]
	Used for operations performed by a client connection. Note that connections are numbered by a sequence number that begins at 1.
Channel	[con:*connectionnumber*(*userid*@/ *clientip*:*ephemeralport*/*virtualhostname*/ ch:*channelnumber*)]
	Used for operations performed by a client's channel (corresponds to the JMS concept of Session). Note that channels are numbered by a sequence number that is scoped by the owning connection.
Group	[grp(/*groupname*)/vhn(/*virtualhostnode name*)]
	Used for HA. Used for operations performed by the system itself often as a result of actions performed on another node..

Table C.2. Subject Entities

Subject Type	Format and Purpose
Connection	[con:*connectionnumber*(*userid*@/ *clientip*:*ephemeralport*/*virtualhostname*)]
	A connection to the Broker.
Channel	[con:*connectionnumber*(*userid*@/ *clientip*:*ephemeralport*/*virtualhostname*/ ch:*channelnumber*)]
	A client's channel within a connection.
Subscription	[sub:*subscriptionnumber*(vh(/ *virtualhostname*)/qu(*queuename*)]
	A subscription to a queue. This corresponds to the JMS concept of a Consumer.
Queue	[vh(/*virtualhostname*)/qu(*queuename*)]
	A queue on a virtualhost
Exchange	[vh(/*virtualhostname*)/ ex(*exchangetype*/*exchangename*)]
	An exchange on a virtualhost
Binding	[vh(/*virtualhostname*)/ ex(*exchangetype*/*exchangename*)/ qu(*queuename*)/rk(*bindingkey*)]
	A binding between a queue and exchange with the giving binding key.

Subject Type	**Format and Purpose**
Message Store	[vh(/*virtualhostname*)/ ms(*messagestorename*)]
	A virtualhost/message store on the Broker.
HA Group	[grp(/*group name*)]
	A HA group

The following tables lists all the operation log messages that can be produced by the Broker, and the describes the circumstances under which each may be seen.

Table C.3. Broker Log Messages

Message Id	**Message Text / Purpose**
BRK-1001	Startup : Version: *version* Build: *build*
	Indicates that the Broker is starting up
BRK-1002	Starting : Listening on *transporttype* port *portnumber*
	Indicates that the Broker has begun listening on a port.
BRK-1003	Shutting down : *transporttype* port *portnumber*
	Indicates that the Broker has stopped listening on a port.
BRK-1004	Qpid Broker Ready
	Indicates that the Broker is ready for normal operations.
BRK-1005	Stopped
	Indicates that the Broker is stopped.
BRK-1006	Using configuration : *file*
	Indicates the name of the configuration store in use by the Broker.
BRK-1007	Using logging configuration : *file*
	Indicates the name of the log configuration file in use by the Broker.
BRK-1008	*delivered|received* : *size* kB/s peak : *size* bytes total
	Statistic - bytes delivered or received by the Broker.
BRK-1009	*delivered|received* : *size* msg/s peak : *size* msgs total
	Statistic - messages delivered or received by the Broker.
BRK-1014	Message flow to disk active : Message memory use *size of all messages* exceeds threshold *threshold size*
	Indicates that the heap memory space occupied by messages has exceeded the threshold so the flow to disk feature has been activated.

Message Id	Message Text / Purpose
BRK-1015	Message flow to disk inactive : Message memory use *size of all messages* within threshold *threshold size*
	Indicates that the heap memory space occupied by messages has fallen below the threshold so the flow to disk feature has been deactivated.
BRK-1016	Fatal error : *root cause* : See log file for more information
	Indicates that broker was shut down due to fatal error.
BRK-1017	Process : PID *process identifier*
	Process identifier (PID) of the Broker process.

Table C.4. Management Log Messages

Message Id	Message Text / Purpose
MNG-1001	*type* Management Startup
	Indicates that a Management plugin is starting up. Currently supported management plugins are JMX and Web.
MNG-1002	Starting : *type* : Listening on *transporttype* port *port*
	Indicates that a Management plugin is listening on the given port.
MNG-1003	Shutting down : *type* : port *port*
	Indicates that a Management plugin is ceasing to listen on the given port.
MNG-1004	*type* Management Ready
	Indicates that a Management plugin is ready for work.
MNG-1005	*type* Management Stopped
	Indicates that a Management plugin is stopped.
MNG-1007	Open : User *username*
	Indicates the opening of a connection to Management has by the given username.
MNG-1008	Close : User *username*
	Indicates the closing of a connection to Management has by the given username.

Table C.5. Virtual Host Log Messages

Message Id	Message Text / Purpose
VHT-1001	Created : *virtualhostname*
	Indicates that a virtualhost has been created.
VHT-1002	Closed

Message Id	Message Text / Purpose
	Indicates that a virtualhost has been closed. This occurs on Broker shutdown.
VHT-1003	*`virtualhostname`* : *`delivered|received`* : *`size`* kB/s peak : *`size`* bytes total
	Statistic - bytes delivered or received by the virtualhost.
VHT-1004	*`virtualhostname`* : *`delivered|received`* : *`size`* msg/s peak : *`size`* msgs total
	Statistic - messages delivered or received by the virtualhost.
VHT-1005	Unexpected fatal error
	Virtualhost has suffered an unexpected fatal error, check the logs for more details.
VHT-1006	Filesystem is over *`size in %`* per cent full, enforcing flow control.
	Indicates that virtual host flow control is activated when the usage of file system containing Virtualhost message store exceeded predefined limit.
VHT-1007	Filesystem is no longer over *`size in %`* per cent full.
	Indicates that virtual host flow control is deactivated when the usage of file system containing Virtualhost message falls under predefined limit.

Table C.6. Queue Log Messages

Message Id	Message Text / Purpose
QUE-1001	Create : Owner: *`owner`* *`AutoDelete`* [*`Durable`*] *`Transient`* Priority: *`numberofpriorities`*
	Indicates that a queue has been created.
QUE-1002	Deleted
	Indicates that a queue has been deleted.
QUE-1003	Overfull : Size : *`size`* bytes, Capacity : *`maximumsize`*
	Indicates that a queue has exceeded its permitted capacity. See Section 9.2.1, “Producer Flow Control” for details.
QUE-1004	Underfull : Size : *`size`* bytes, Resume Capacity : *`resumesize`*
	Indicates that a queue has fallen to its resume capacity. See Section 9.2.1, “Producer Flow Control” for details.

Table C.7. Exchange Log Messages

Message Id	Message Text / Purpose
EXH-1001	Create : [*`Durable`*] Type: *`type`* Name: *`exchange name`*
	Indicates that an exchange has been created.

Message Id	**Message Text / Purpose**
EXH-1002	Deleted
	Indicates that an exchange has been deleted.
EXH-1003	Discarded Message : Name: *`exchange name`* Routing Key: *`routing key`*
	Indicates that an exchange received a message that could not be routed to at least one queue. queue has exceeded its permitted capacity. See Section 4.6.4, “Unrouteable Messages” for details.

Table C.8. Binding Log Messages

Message Id	**Message Text / Purpose**
BND-1001	Create : Arguments : *`arguments`*
	Indicates that a binding has been made between an exchange and a queue.
BND-1002	Deleted
	Indicates that a binding has been deleted

Table C.9. Connection Log Messages

Message Id	**Message Text / Purpose**
CON-1001	Open : Client ID : *`clientid`* : Protocol Version : *`protocol version`* : Client Version : *`client version`* : Client Product :*`client product`*
	Indicates that a connection has been opened. The Broker logs one of these message each time it learns more about the client as the connection is negotiated.
CON-1002	Close
	Indicates that a connection has been closed. This message is logged regardless of if the connection is closed normally, or if the connection is somehow lost e.g network error.
CON-1003	Closed due to inactivity
	Used when heart beating is in-use. Indicates that the connection has not received a heartbeat for too long and is therefore closed as being inactive.

Table C.10. Channel Log Messages

Message Id	**Message Text / Purpose**
CHN-1001	Create
	Indicates that a channel (corresponds to the JMS concept of Session) has been created.
CHN-1002	Flow Started
	Indicates message flow to a session has begun.
CHN-1003	Close

Message Id	**Message Text / Purpose**
	Indicates that a channel has been closed.
CHN-1004	Prefetch Size (bytes) `size` : Count `number of messages`
	Indicates the prefetch size in use by a channel.
CHN-1005	Flow Control Enforced (Queue `queue name`)
	Indicates that producer flow control has been imposed on a channel owning to excessive queue depth in the indicated queue. Produces using the channel will be requested to pause the sending of messages. See Section 9.2.1, “Producer Flow Control” for more details.
CHN-1006	Flow Control Removed
	Indicates that producer flow control has been removed from a channel. See Section 9.2.1, “Producer Flow Control” for more details.
CHN-1007	Open Transaction : `time` ms
	Indicates that a producer transaction has been open for longer than that permitted. See Section 9.3, “Producer Transaction Timeout” for more details.
CHN-1008	Idle Transaction : `time` ms
	Indicates that a producer transaction has been idle for longer than that permitted. See Section 9.3, “Producer Transaction Timeout” for more details.
CHN-1009	Discarded message : `message number` as no alternate exchange configured for queue : `queue name`{1} routing key : `routing key`
	Indicates that a channel has discarded a message as the maximum delivery count has been exceeded but the queue defines no alternate exchange. See Section 9.4.2, “Maximum Delivery Count” for more details. Note that `message number` is an internal message reference.
CHN-1010	Discarded message : `message number` as no binding on alternate exchange : `exchange name`
	Indicates that a channel has discarded a message as the maximum delivery count has been exceeded but the queue's alternate exchange has no binding to a queue. See Section 9.4.2, “Maximum Delivery Count” for more details. Note that `message number` is an internal message reference.
CHN-1011	Message : `message number` moved to dead letter queue : `queue name`
	Indicates that a channel has moved a message to the named dead letter queue
CHN-1012	Flow Control Ignored. Channel will be closed.
	Indicates that a channel violating the imposed flow control has been closed

Message Id	Message Text / Purpose
CHN-1013	Uncommitted transaction contains *size* bytes of incoming message data.
	Warns about uncommitted transaction with large message(s)

Table C.11. Subscription Log Messages

Message Id	Message Text / Purpose
SUB-1001	Create : [*Durable*] Arguments : *arguments*
	Indicates that a subscription (corresponds to JMS concept of a MessageConsumer) has been created.
SUB-1002	Close
	Indicates that a subscription has been closed.
SUB-1003	SUB-1003 : Suspended for *time* ms
	Indicates that a subscription has been in a suspened state for an unusual length of time. This may be indicative of an consuming application that has stopped taking messages from the consumer (i.e. a JMS application is not calling receive() or its asynchronous message listener onMessage() is block in application code). It may also indicate a generally overloaded system.

Table C.12. Message Store Log Messages

Message Id	Message Text / Purpose
MST-1001	Created
	Indicates that a message store has been created. The message store is responsible for the storage of the messages themselves, including the message body and any headers.
MST-1002	Store location : *path*
	Indicates that the message store is using *path* for the location of the message store.
MST-1003	Closed
	Indicates that the message store has been closed.
MST-1004	Recovery Start
	Indicates that message recovery has begun.
MST-1005	Recovered *number of messages* messages.
	Indicates that recovery recovered the given number of messages from the store.
MST-1006	Recovered Complete
	Indicates that the message recovery is concluded.
MST-1007	Store Passivated

Message Id	Message Text / Purpose
	The store is entering a passive state where is it unavailable for normal operations. Currently this message is used by HA when the node is in replica state.
MST-1008	Store overfull, flow control will be enforced
	The store has breached is maximum configured size. See Section 9.2.1, “Producer Flow Control” for details.
MST-1009	Store overfull condition cleared
	The store size has fallen beneath its resume capacity and therefore flow control has been rescinded. See Section 9.2.1, “Producer Flow Control” for details.

Table C.13. Transaction Store Log Messages

Message Id	Message Text / Purpose
TXN-1001	Created
	Indicates that a transaction store has been created. The transaction store is responsible for the storage of messages instances, that is, the presence of a message on a queue.
TXN-1002	Store location : *`path`*
	Indicates that the transaction store is using *`path`* for the location of the store.
TXN-1003	Closed
	Indicates that the transaction store has been closed.
TXN-1004	Recovery Start
	Indicates that transaction recovery has begun.
TXN-1005	Recovered *`number`* messages for queue *`name`*.
	Indicates that recovery recovered the given number of message instances for the given queue.
TXN-1006	Recovered Complete
	Indicates that the message recovery is concluded.

Table C.14. Configuration Store Log Messages

Message Id	Message Text / Purpose
CFG-1001	Created
	Indicates that a configuration store has been created. The configuration store is responsible for the storage of the definition of objects such as queues, exchanges, and bindings.
CFG-1002	Store location : *`path`*
	Indicates that the configuration store is using *`path`* for the location of the store.
CFG-1003	Closed
	Indicates that the configuration store has been closed.

Message Id	Message Text / Purpose
CFG-1004	Recovery Start
	Indicates that configuration recovery has begun.
CFG-1005	Recovered Complete
	Indicates that the configuration recovery is concluded.

Table C.15. HA Log Messages

Message Id	Message Text / Purpose
HA-1001	Created
	This HA node has been created.
HA-1002	Deleted
	This HA node has been deleted
HA-1003	Added : Node : '`name`' (`host:port`)
	A new node has been added to the group.
HA-1004	Removed : Node : '`name`' (`host:port`)
	The node has been removed from the group. This removal is permanent.
HA-1005	Joined : Node : '`name`' (`host:port`)
	The node has become reachable. This may be as a result of the node being restarted, or a network problem may have been resolved.
HA-1006	Left : Node : '`name`' (`host:port`)
	The node is no longer reachable. This may be as a result of the node being stopped or a network partition may be preventing it from being connected. The node is still a member of the group.
HA-1007	HA-1007 : Master transfer requested : to '`name`' (`host:port`)
	Indicates that a master transfer operation has been requested.
HA-1008	HA-1008 : Intruder detected : Node '`name`' (`host:port`)
	Indicates that an unexpected node has joined the group. The virtualhost node will go into the ERROR state in response to the condition.
HA-1009	HA-1009 : Insufficient replicas contactable
	This node (which was in the master role) no longer has sufficient replica in contact in order to complete transactions.
HA-1010	HA-1010 : Role change reported: Node : '`name`' (`host:port`) : from `role` to `role`
	Indicates that the node has changed role within the group.
HA-1011	HA-1011 : Minimum group size : `new group size`

Message Id	Message Text / Purpose
	The quorum requirements from completing elections or transactions has been changed.
HA-1012	HA-1012 : Priority : `priority`
	The priority of the object node has been changed. Zero indicates that the node cannot be elected master.
HA-1013	HA-1013 : Designated primary : `true|false`
	This node has been designated primary and can now operate solo. Applies to two node groups only.
HA-1014	HA-1014 : Diverged transactions discarded
	This node is in the process of rejoining the group but has discovered that some of its transactions differ from those of the current master. The node will automatically roll-back (i.e. discard) the diverging transactions in order to be allowed to rejoin the group. This situation can only usually occur as a result of use of the weak durability options. These allow the group to operate with fewer than quorum nodes and therefore allow the inconsistencies to develop. On encountering this condition, it is *strongly* recommendend to run an application level reconcilation to determine the data that has been lost.

Table C.16. Port Log Messages

Message Id	Message Text / Purpose
PRT-1001	Create
	Port has been created.
PRT-1002	Open
	Port has been open
PRT-1003	Close
	Port has been closed
PRT-1004	Connection count `number` within `warn limit` % of maximum `limit`
	Warns that number of open connections approaches maximum allowed limit
PRT-1005	Connection from `host` rejected
	Connection from given host is rejected because of reaching the maximum allowed limit

Appendix D. Queue Alerts

The Broker supports a variety of queue alerting thresholds. Once configured on a queue, these limits will be periodically written to the log if these limits are breached, until the condition is rectified.

For example, if queue `myqueue` is configured with a message count alert of 1000, and then owing to a failure of a downstream system messages begin to accumulate on the queue, the following alerts will be written periodically to the log.

```
INFO [default:VirtualHostHouseKeepingTask] (queue.NotificationCheck) - MESSAGE_COU
           On Queue myqueue - 1272: Maximum count on queue threshold (1000) breach
```

Note that queue alerts are *soft* in nature; breaching the limit will merely cause the alerts to be generated but messages will still be accepted to the queue.

Table D.1. Queue Alerts

Alert Name	Alert Format and Purpose
MESSAGE_COUNT_ALERT	MESSAGE_COUNT_ALERT On Queue *`queuename`* - *`number of messages`*: Maximum count on queue threshold (*`limit`*) breached.
	The number of messages on the given queue has breached its configured limit.
MESSAGE_SIZE_ALERT	MESSAGE_SIZE_ALERT On Queue *`queuename`* - *`message size`* : Maximum message size threshold (*`limit`*) breached. [Message ID=*`message id`*]
	The size of an individual messages has breached its configured limit.
QUEUE_DEPTH_ALERT	QUEUE_DEPTH_ALERT On Queue *`queuename`* - *`total size of all messages on queue`* : Maximum queue depth threshold (*`limit`*) breached.
	The total size of all messages on the queue has breached its configured limit.
MESSAGE_AGE_ALERT	MESSAGE_AGE_ALERT On Queue *`queuename`* - *`age of message`* : Maximum age on queue threshold (*`limit`*) breached.
	The age of a message on the given queue has breached its configured limit.

Appendix E. Miscellaneous

E.1. JVM Installation verification

E.1.1. Verify JVM on Windows

Firstly confirm that the JAVA_HOME environment variable is set correctly by typing the following at the command prompt:

```
echo %JAVA_HOME%
```

If JAVA_HOME is set you will see something similar to the following:

```
c:"\PROGRA~1"\Java\jdk1.7.0_60\
```

Then confirm that a Java installation (1.7 or higher) is available:

```
java -version
```

If java is available on the path, output similar to the following will be seen:

```
java version "1.7.0_60"
Java(TM) SE Runtime Environment (build 1.7.0_60-b19)
Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
```

E.1.2. Verify JVM on Unix

Firstly confirm that the JAVA_HOME environment variable is set correctly by typing the following at the command prompt:

```
echo $JAVA_HOME
```

If JAVA_HOME is set you will see something similar to the following:

```
/usr/java/jdk1.7.0_60
```

Then confirm that a Java installation (1.7 or higher) is available:

```
java -version
```

If java is available on the path, output similar to the following will be seen:

```
java version "1.7.0_60"
Java(TM) SE Runtime Environment (build 1.7.0_60-b19)
Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
```

E.2. Installing External JDBC Driver

In order to use a JDBC Virtualhost Node or a JDBC Virtualhost, you must make the Database's JDBC 4.0 compatible drivers available on the Broker's classpath. To do this copy the driver's JAR file into the `${QPID_HOME}/lib/opt` folder.

```
Unix:
cp driver.jar qpid-broker-0.32-SNAPSHOT/lib/opt

Windows:
copy driver.jar qpid-broker-0.32-SNAPSHOT\lib\opt
```

E.3. Installing Oracle BDB JE

The Oracle BDB JE is not distributed with Apache Qpid owing to license considerations..

If you wish to use a BDB Virtualhost Node, BDB Virtualhost, or BDB HA Virtualhost Node you must make the BDB JE's JAR available on the Broker's classpath.

Download the Oracle BDB JE 5.0.104 release from the Oracle website. [http://www.oracle.com/technetwork/products/berkeleydb/downloads/index.html?ssSourceSiteId=ocomen]

The download has a name in the form je-5.0.104.tar.gz. It is recommended that you confirm the integrity of the download by verifying the MD5.

Copy the je-5.0.104.jar from within the release into `${QPID_HOME}/lib/opt` folder.

```
Unix:
cp je-5.0.104.jar qpid-broker-0.32-SNAPSHOT/lib/opt

Windows:
copy je-5.0.104.jar qpid-broker-0.32-SNAPSHOT\lib\opt
```