Apache Qpid Broker-J

Apache Qpid Broker-J

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

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

This document relates to the Apache Qpid Broker-J. The C++ Broker is described separately [/releases/qpid-cpp-{{current_cpp_release}}/cpp-broker/book/].

Headline features

- 100% Java implementation - runs on any platform supporting Java 11 or higher
- Messaging clients support in Java (JMS 1.1 and 2.0 compliance), C++, Python and more.
- 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 (0-8, 0-9, 0-9-1, 0-10 and 1.0).
- Supports for AMQP over websockets.
- 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, OAuth2, and file-based authentication mechanisms.
- Support for message compression
- Pluggable storage architecture with implementations including Apache Derby [http://db.apache.org/derby/], Oracle BDB JE [${oracleBdbProductOverviewUrl}], and External Databases.
- Web based management interface and programmatic management interfaces via REST.
- SSL support
- High availability (HA) support.[2]

[1] XA provided when using AMQP 0-10
[2] HA currently only available with the Oracle BDB JE message store option.

Chapter 2. Installation

2.1. Introduction

This document describes how to install the Apache Qpid Broker-J on both Windows and UNIX platforms.

2.2. Prerequisites

2.2.1. Java Platform

The Apache Qpid Broker-J is an 100% Java implementation and as such it can be used on any operating system supporting Java 11 or higher[1]. This includes Linux, Solaris, Mac OS X, and Windows 7/8/10 etc.

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 Broker installation requires approximately 20MB of free disk space.

The 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 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 in memory for performance reasons, so in general, the Broker will benefit from as much memory 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 usable size of approximately ~3.7GB.

See Section 9.11, “Memory” for a full description of how memory is used.

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 Broker.

[1] Java Cryptography Extension (JCE) Unlimited Strength extension must be installed or enabled for some features.

2.3. Download

2.3.1. Broker Release

You can download the latest 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-broker-9.2.0-bin.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\\9.2.0 within c:\qpid

```
 Directory of c:\qpid\qpid-broker\\9.2.0

25/11/2015  11:29    <DIR>          .
25/11/2015  11:29    <DIR>          ..
25/11/2015  10:56    <DIR>          bin
03/07/2015  08:06    <DIR>          etc
25/11/2015  11:25    <DIR>          lib
25/11/2015  10:56            28,143 LICENSE
25/11/2015  10:56             3,409 NOTICE
29/04/2015  09:13               116 README.txt
3 File(s)         31,668 bytes
5 Dir(s)  25,981,767,680 bytes free
```

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 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-broker-9.2.0-bin.tgz package into the directory.

```
mkdir /usr/local/qpid
cd /usr/local/qpid
tar xvzf qpid-broker-9.2.0-bin.tgz
```

The extraction of the broker package will have created a directory qpid-broker/9.2.0 within /usr/local/qpid

```
ls -la qpid-broker/9.2.0/
total 56
drwxrwxr-x. 5 alex alex  4096 Nov 25 11:43 .
drwxrwxr-x. 3 alex alex  4096 Nov 25 11:43 ..
drwxr-xr-x. 2 alex alex  4096 Nov 24 23:38 bin
drwxr-xr-x. 2 alex alex  4096 Nov 24 23:38 etc
drwxrwxr-x. 2 alex alex  4096 Nov 25 11:43 lib
-rw-r--r--. 1 alex alex 28143 Nov 24 23:38 LICENSE
-rw-r--r--. 1 alex alex  3409 Nov 24 23:38 NOTICE
-rw-r--r--. 1 alex alex   116 Nov 24 23:38 README.txt
```

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 Broker will attempt to create it on start-up.

2.6. Optional Dependencies

If you wish to utilise a storage option using an External Database, see Section F.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 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-1001 : Startup : Version: 9.2.0 Build: 1478262
[Broker] BRK-1010 : Platform : JVM : Oracle Corporation version: 11.0.13+10-LTS-37
[Broker] BRK-1011 : Maximum Memory : Heap : 518,979,584 bytes Direct : 1,610,612,7
[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] 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 on which the Broker is listening (for HTTP management and AMQP respectively).

To stop the Broker, use Control-C from the controlling command prompt or REST operation broker/shutdown.

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-1001 : Startup : Version: 9.2.0 Build: exported
[Broker] BRK-1010 : Platform : JVM : Oracle Corporation version: 11.0.13+10-LTS-37
[Broker] BRK-1011 : Maximum Memory : Heap : 518,979,584 bytes Direct : 1,610,612,7
```

```
[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] 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 on which the Broker is listening (for HTTP 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 REST operation broker/shutdown.

3.4. Log file

The 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, “Logging”

3.5. Using the command line

The 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>] [-mm] [-mmhttp <port>]
       [-mmpass <password>] [-mmqv] [-os]
       [-prop <name=value>] [-props <path>] [-sp <path>] [-st <type>] [-v]
 -cic,--create-initial-config <path>                  create a copy of the
                                                      initial config file,
                                                      either to an
                                                      optionally specified
                                                      file path, or as
                                                      initial-config.json
                                                      in the current
                                                      directory
 -h,--help                                            print this message
 -icp,--initial-config-path <path>                    set the location of
                                                      initial JSON config
                                                      to use when
                                                      creating/overwriting
                                                      a broker
                                                      configuration store
 -mm,--management-mode                                start broker in
                                                      management mode,
                                                      disabling the AMQP
```

```
                                                    ports
-mmhttp,--management-mode-http-port <port>          override http
                                                    management port in
                                                    management mode
-mmpass,--management-mode-password <password>       Set the password for
                                                    the management mode
                                                    user mm_admin
-mmqv,--management-mode-quiesce-virtualhostnodes    make virtualhost nodes
                                                    stay in the quiesced
                                                    state during
                                                    management mode.
-prop,--config-property <name=value>                set a configuration
                                                    property to use when
                                                    resolving variables
                                                    in the broker
                                                    configuration store,
                                                    with format
                                                    "name=value"
-props,--system-properties-file <path>              set the location of
                                                    initial properties
                                                    file to set
                                                    otherwise unset
                                                    system properties
-sp,--store-path <path>                             use given
                                                    configuration store
                                                    location
-st,--store-type <type>                             use given broker
                                                    configuration store
                                                    type
-v,--version                                        print the version
                                                    information and exit
```

Chapter 4. Concepts

4.1. Overview

The Broker comprises 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 container in which messaging is performed. A *virtualhost* exists in a container called a *virtualhost node*. A virtualhost node has exactly one virtualhost.

An *Exchange* accepts messages from a producer application and routes these to one or more *Queues* according to pre-arranged criteria called *bindings*. Exchange are an AMQP 0-8, 0-9, 0-9-1, 0-10 concept. They exist to produce useful messaging behaviours such as fanout. When using AMQP 0-8, 0-9, 0-9-1, or 0-10, the exchange is the only way ingressing a message into the virtualhost. When using AMQP 1.0, the application may route messages using an exchange (to take advantage of advanced behaviours) or it may publish messages direct to a queue.

*Queue*s are named entities that hold/buffer messages for later delivery to consumer applications.

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.

Connection Limit Provider restrains users (or groups of users) at opening new connections on AMQP ports.

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.

Loggers, at this point in the hierarchy, are responsible for the production of a log for the Broker.

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 showing major entities

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.

Virtualhost node also provides an initial configuration for its *virtualhost*. How to specify initial configuration for *virtual host* is described at Section 5.8, “Virtualhost Initial Configuration”.

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 container 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 virtualhost node.

The virtualhost comprises 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. When using AMQP 0-8, 0-9, 0-9-1, 0-10 the exchange is the only way ingressing a message into the virtualhost. When using AMQP 1.0 producers may route messages via exchanges or direct to queues.

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

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.

Loggers are responsible for producing logs for this virtualhost.

The following diagram depicts the Virtualhost model:

Figure 4.3. Virtualhost Model showing major entities

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*.

When using AMQP 0-8, 0-9, 0-9-1, or 0-10, the exchange is the only way ingressing a message into the virtualhost. When using AMQP 1.0, the application may route messages using an exchange (to take advantage of exchange's routing behaviours), or it may route direcly to a queue (if point to point messaging is required).

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)

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 the AMQP 1.0 protocol, and an alternate binding has been set on the exchange, the message is routed to the alternate. If the message is still unroutable after considering the alternate binding, the message is discarded unless the sending link has requested the `REJECT_UNROUTABLE` target capability, or the Exchange has its `unroutableMessageBehaviour` attribute set to `REJECT`.
- If using the AMQP 0-10 protocol, and an alternate binding has been set on the exchange, the message is routed to the alternate. If the message is still unroutable after considering the alternate binding,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.

Messages arrive on queues either from Exchanges, or when using the AMQP 1.0 protocol, the producing application can direct messages straight to the queue. For AMQP 0-8, 0-9, 0-9-1, or 0-10, the exchange is the only way ingressing a message into a queue.

Consumers subscribe to a queue in order to receive messages from it.

[1] This is a Qpid specific extension.

The Broker supports different queue types, each with different delivery semantics. Queues also have a range of other features such as the ability to group messages together for delivery to a single consumer. These additional features are described below too.

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 put at the head of 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. 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.

4.7.2.1. Grouping Messages

In order to group messages, JMS applications can set the JMS standard header `JMSXGroupId` to specify the *group identifier* when publishing messages.

Alternatively, the application may 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.

4.7.2.2. 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 guarantees 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.

4.7.3. 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.

4.7.3.1. 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. The attribute names are `minimumMessageTtl` and `maximumMessageTtl` and the TTL value is given in milliseconds.

4.7.3.2. 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.

When using the Qpid JMS AMQP 0-x, the consumer declaration can be hinted using the address.

Table 4.1. Setting the replay period using a Qpid JMS AMQP 0-x address

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'

4.7.3.3. 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" ] } }
```

4.7.4. Holding messages on a Queue

Sometimes it is required that while a message has been placed on a queue, it is not released to consumers until some external condition is met.

4.7.4.1. Hold until valid

Currently Queues support the "holding" of messages until a (per-message) provided point in time. By default this support is not enabled (since it requires extra work to be performed against every message entering the queue. To enable support, the attribute `holdOnPublishEnabled` must evaluate to true for

the Queue. When enabled messages on the queue will be checked for the header (for AMQP 0-8, 0-9, 0-9-1 and 0-10 messages) or message annotation (for AMQP 1.0 messages) `x-qpid-not-valid-before`. If this header/annotation exists and contains a numeric value, it will be treated as a point in time given in milliseconds since the UNIX epoch. The message will not be released from the Queue to consumers until this time has been reached.

4.7.5. Controlling Queue Size

Overflow Policy can be configured on an individual *Queue* to limit the queue size. The size can be expressed in terms of a *maximum number of bytes* and/or *maximum number of messages*. The *Overflow Policy* defines the Queue behaviour when any of the limits is reached.

The following *Overflow Policies* are supported:

- *None* - Queue is unbounded and the capacity limits are not applied. This is a default policy applied implicitly when policy is not set explicitly.
- *Ring* - If a newly arriving message takes the queue over a limit, message(s) are deleted from the queue until the queue falls within its limit again. When deleting messages, the oldest messages are deleted first. For a Priority Queue the oldest messages with lowest priorities are removed.
- *Producer Flow Control* -The producing sessions are blocked until queue depth falls below the *resume threshold* set as a context variable *${queue.queueFlowResumeLimit}* (specifying the percentage from the limit values. Default is 80%).
- *Flow to Disk* -If the queue breaches a limit, newly arriving messages are written to disk and the in-memory representation of the message is minimised. The Broker will transparently retrieve messages from disk as they are required by a consumer or management. The flow to disk policy does not actually restrict the overall size of the queue, merely the space occupied in memory. The Broker's other Flow to Disk feature operates completely independent of this Queue Overflow Policy.
- *Reject* -A newly arriving message is rejected when queue limit is breached.

A negative value for *maximum number of messages* or *maximum number of bytes* disables the limit.

The Broker issues Operational log messages when the queue sizes are breached. These are documented at Table C.6, “Queue Log Messages”.

4.7.6. Using low pre-fetch with special queue types

Messaging clients may buffered messages for performance reasons. In Qpid, this is commonly known as *pre-fetch*

When using some of the messaging features described on this section, using prefetch can give unexpected behaviour. 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.

Using pre-fetch of 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.

See the messaging client documentation for details of how to configure prefetch.

4.8. Ports

The Broker supports configuration of *Ports* to specify the particular AMQP messaging and HTTP 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 ports can be configured for use by the associated management plugin.

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*, *Connection Limit 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 Providers”.

4.10.2. Connection Limit Providers

Connection Limit Providers are used to limit the amount of connections that user could open on AMQP ports.

Connection Limit Providers configuration and management details are covered in Section 8.4, “Connection Limit Providers”.

4.10.3. 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”.

4.10.4. Keystores

Keystores are used to configure SSL private and public keys and certificates for the SSL transports on Ports.

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

4.10.5. Truststores

Truststores are used to configure SSL certificates for trusting Client Certificate on SSL ports or making SSL connections to other external services like LDAP, etc.

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

4.10.6. Loggers

Loggers are responsible for producing a log of events from either the Broker as a whole, or an individual Virtualhost. These are described in Section 9.1, “Logging”.

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 Nodes
- Loggers
- Ports
- Authentication Providers (optionally with Users for managing users Authentication Providers)
- Access Control Providers
- User Connection Limit Providers
- Group Providers (optionally with Groups and GroupMembers for managing groups Group Providers)
- Key stores
- Trust stores
- Plugins

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'. 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.

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 Broker's 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 5.7, “Example of JSON 'Initial Configuration'”

5.5. Configuration Store Type

There are currently several implementations of the pluggable Broker Configuration Store:

JSON	the default one which persists content to disk in a JSON file
Memory	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.
DERBY	stores configuration in embedded derby store
BDB	stores configuration in Berkeley DB store
JDBC	stores configuration in external RDBMS using JDBC

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'

Another example of broker startup with configuration in DERBY network server

```
$ ./qpid-server -st JDBC \
  -prop "systemConfig.connectionUrl=jdbc:derby://localhost:1527/path/to/store;crea
  -prop "systemConfig.username=test" -prop "systemConfig.password=password"
```

5.6. 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 ConfigurationProperties 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.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 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 at first broker startup 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.

5.7. 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": "${broker.name}",
  "modelVersion" : "9.0",
    "authenticationproviders" : [ {
      "name" : "plain",
      "type" : "Plain",
      "users" : [ {
      "name" : "guest",
      "type" : "managed",
      "password" : "guest"
      } ]
    } ],
    "brokerloggers" : [ {
    "name" : "logfile",
    "type" : "File",
    "fileName" : "${qpid.work_dir}${file.separator}log${file.separator}qpid.log",
    "brokerloginclusionrules" : [ {
      "name" : "Root",
      "type" : "NameAndLevel",
      "level" : "WARN",
      "loggerName" : "ROOT"
    }, {
      "name" : "Qpid",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "org.apache.qpid.*"
    }, {
      "name" : "Operational",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "qpid.message.*"
    }, {
      "name" : "Statistics",
      "type" : "NameAndLevel",
      "level" : "INFO",
```

```
      "loggerName" : "qpid.statistics.*"
    } ]
  }, {
    "name" : "memory",
    "type" : "Memory",
    "brokerloginclusionrules" : [ {
      "name" : "Root",
      "type" : "NameAndLevel",
      "level" : "WARN",
      "loggerName" : "ROOT"
    }, {
      "name" : "Qpid",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "org.apache.qpid.*"
    }, {
      "name" : "Operational",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "qpid.message.*"
    }, {
      "name" : "Statistics",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "qpid.statistics.*"
    } ]
  } ],
  "ports" : [  {
    "name" : "AMQP",
    "port" : "${qpid.amqp_port}",
    "authenticationProvider" : "plain",
    "virtualhostaliases" : [ {
       "name" : "nameAlias",
       "type" : "nameAlias"
    }, {
        "name" : "defaultAlias",
        "type" : "defaultAlias"
    }, {
        "name" : "hostnameAlias",
        "type" : "hostnameAlias"
    } ]
  }, {
    "name" : "HTTP",
    "port" : "${qpid.http_port}",
    "authenticationProvider" : "plain",
    "protocols" : [ "HTTP" ]
  }],
  "virtualhostnodes" : [ {
    "name" : "default",
    "type" : "JSON",
    "defaultVirtualHostNode" : "true",
    "virtualHostInitialConfiguration" : "\\${qpid.initial_config_virtualhost_confi
  } ],
  "plugins" : [ {
```

```
    "type" : "MANAGEMENT-HTTP",
    "name" : "httpManagement"
  } ]
}
```

In the configuration above the following entries are stored:

- Authentication Provider of type *PlainPasswordFile* with name "passwordFile".
- Two Port entries: "AMQP", "HTTP"
- Virtualhost Node called *default*.
- One management plugin: "httpManagement" of type "MANAGEMENT-HTTP".
- Broker attributes are stored as a root entry.

5.8. Virtualhost Initial Configuration

Virtualhost initial configuration can be specified in *Virtualhost node* attribute *virtualHostInitialConfiguration*. On first startup, the *virtualhost* is created based on provided initial configuration. You can define there manageable *Virtualhost* attributes and children like exchanges, queues, etc.

The attribute `virtualHostInitialConfiguration` can have a value of *URL* to an external resource where *virtualhost* initial configuration is provided in json format, or, it can hold a string value with initial configuration in stringified json format. If required, you can specify initial configuration as context variable which can be resolved as *URL* to external resource or stringified json.

Example 5.2. Example of virtual host initial configuration provided as stringified JSON

```
...
"virtualhostnodes" : [ {
"name" : "default",
"type" : "JSON",
"defaultVirtualHostNode" : "true",
"virtualHostInitialConfiguration" : "{\"type\":\"BDB\",\"nodeAutoCreat
} ]
...
```

After creation of *virtualhost* the value of `virtualHostInitialConfiguration` is set to an empty string.

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 a Web Console and a 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

6.3.1. Introduction

This section describes the REST API provided by the Apache Qpid Broker-J. The REST API is intended for use by developers who wish to automate the management or monitoring of the Broker. It is also very useful for adhoc monitoring on the command line using tools such as `curl`.

The REST API provides access to all of the Broker's entities using hierarchical paths expressed by the URI. Responses are returned in JSON format.

The `GET` method request retrieves information about an object, the `DELETE` method requests the removal of one, and the `PUT` or `POST` methods perform updates or create new objects. The `POST` method is also used to invoke operations.

The REST API is versioned with the version number embedded within the URI. The general form of the URI is `/api/<version>` where <version> is a dot separated major and minor model version prefixed with "v", for example, "v6.1" (without the quotation marks). For convenience the alias `latest` (`/api/latest`) signifies the latest supported version.

There are also some ancillary services under URI `/service` used for authentication and logout.

6.3.2. REST API documentation

REST API documentation is available on-line from any Broker at location `/apidocs`. It is also linked from the menu of the Web Management Console.

6.3.3. Authentication

Before you can use the REST API, you must authenticate. Authentication decisions are made by the authentication provider associated with HTTP port on which you connect.

You may authenticate using SASL [https://www.ietf.org/rfc/rfc4422.txt] (`/service/sasl`) or HTTP Basic Authentication [https://tools.ietf.org/html/rfc2617]. The latter is convienent when using tools such as `curl` on the command line. This is illustrated in the examples below.

For SASL authentication use a `GET` request to `/service/sasl` to get a list of supported SASL mechanisms, and use `PUT` to the same URL to perform the SASL negotiation.

It is possible to end an authenticated session using `/service/logout`.

6.3.4. 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.

6.3.5. 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
```

6.3.6. 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
```

6.3.7. Retrieving Configured Object details

Method GET is used to retrieve an object's attributes values and statistics.

To retrieve a single object, use its full URI. For instance, to retrieve a single queue:

```
GET /api/latest/queue/vhn/vh/my-queue
```

To retrieve all objects beneath a parent, pass the parent's URI. For instance, to retrieve all queues beneath the virtualhost called `vh`. A collection will be returned.

```
GET /api/latest/queue/vhn/vh
```

Request parameters (with the same name as an attribute) are used to filter the returned collection. For instance, to filter those queues of type `standard`:

```
GET /api/latest/queue/vhn/vh?type=standard
```

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
excludeInheritedContext	If set to "false" the inherited context is included from the object's ancestors. Default is true.
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.

6.3.8. Configured Object operations

Method POST is used to invoke Configured Objects operations. Some operations support parameters. Pass parameters using a JSON request body containing a map with a map entry for each parameter.

Example 6.4. Example REST call invoking the operation clear queue

To clear the queue with name "my-queue" on a virtual host with name "vh".

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

6.3.9. HTTP status codes returned by REST interfaces

Table 6.1. 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.
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

Status code	Description
	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.

6.3.10. Examples of REST requests with curl

Example 6.5. 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
```

NOTE: These curl examples utilise an unsecured 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 7.17, “HTTP Plugin”

6.3.11. Query API

6.3.11.1. Introduction

The *Qpid Broker-J* provides a powerful feature called the *Query API*. This allows the retrieval of the existing configured objects attributes satisfying user-provided queries.

Developers and operators can use this feature to monitor the Broker. For example, using *Query API* one can find all queues with queue depth exceeding some limit or existing connections made from a particular location(s).

6.3.11.2. Query API Overview

When using the *Query API* one specifies the category of the object to query, a list of attributes to return in the result set, an optional where clause, expressed as a predicate, that determines the filtering criteria, ordering, and limit/offset. The features should be readily recognisable to anyone who has has familiarity with SQL.

Queries associate with either the *broker* as a whole, or an individual *virtualhost*. Queries associated with the Broker can query any object within the Broker. Queries associated with a virtualhost are limited to the objects of the virtualhost itself. For instance a queue query associated with a virtualhost queries only the queues belonging to that virtualhost. On the other hand, a queue query associated with the Broker sees all the queues belonging on the entire Broker.

Table 6.2. Query API URLs

Query API URL	Description
/api/latest/querybroker/<configured object category name> /api/<version>/querybroker/<configured object category name>	Query API URL fragment to query the specified object type across the entire broker
/api/latest/queryvhost/<virtual host node name>/<virtual host name>/<configured object category name> /api/<version>/queryvhost/<virtual host node name>/<virtual host name>/<configured object category name>	Query API URL fragment to query the specified object type for a specific virtualhost

The QueryAPI accepts `select`, `where`, `orderBy`, `limit` and `offset` request parameters.

Table 6.3. Query API request parameters

Parameter Name	Parameter Description
`select`	The `select` defines the columns of the result set. It is a comma-separated list of expressions. At its most simple, an expression can be the name of the attribute (e.g. `queueDepthBytes`), but more complex expressions are also supported. Columns within the result set are named. For expressions that are simple attribute names, the column names will follow the attributes themselves. By default, other expressions will have a no name. Column names can be overridden with an `AS` clause e.g. `now() AS currentDate`
`where`	The `where` provides a boolean expression defining the result set filtering. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax and is similar to selector expressions in JMS e.g.

Parameter Name	Parameter Description
	`queueDepthBytes > 16384 AND name like '%flow_queue'`.
`orderBy`	Ordering conditions; the syntax of the expression is based on a subset of the SQL92 ordering expression syntax. Similar to ordering expressions in SQL, one can specify in ordering expression attributes names, sub-expressions or indexes (starting from 1) of attributes or expressions specified in select.
`limit`	The maximum number of results to provide starting from given offset.
`offset`	An offset in results (default is 0) to provide results from.

Example 6.6. Example of a Query API request to retrieve queue names and depths.

```
GET api/latest/querybroker/queue?select=name,queueDepthBytes,queueDepthMessages&wh
```

6.3.11.3. Query API Results

The *Query API* returns a JSON response. The response contains the following:

`headers` ordered list of result set column names derived from the `select` clause. Note that anonymous expressions (that is, those expressed without an `AS`) will have empty column name.

`results` two dimensional array containing the result-set

`total` The *total* number of results matching the where criteria.

Example 6.7. Example of Query API call for queue names and depths.

```
GET api/latest/querybroker/queue?select=name,queueDepthBytes,queueDepthMessages&wh
```

```
{
    "headers" : [ "name", "queueDepthBytes", "queueDepthMessages" ],
    "results" : [ [ "foo", 312, 26], [ "bar", 300, 24 ] ],
    "total" : 2
}
```

Query API expressions

Expressions within the `select`, `where` and `orderBy` clauses can be comprised in the following manner. Expressions can be nested to arbitary depth. Parentheses allow for precedence to be explicitly denoted.

- variable name which can be an attribute name e.g `queueDepthBytes` or a reference to a parent attribute `$parent.name`
- literal e.g. `3` or `'foo'`

- functions - see below e.g. `now()` or `to_string(createdDate, '%tm/%td/%ty', 'EST')`
- arithmetic operations e.g. `3 * 4` or `to_string(now()) + name`

The following functions are supported:

Table 6.4. Query API functions

Function Name	Function Description
`concat(obj[,obj..])`	concatenates the given objects into a string
`now()`	returns current date and time
`to_date(object)`	converts the first parameter, which must be a string. into a date. The string must be in ISO-8601 format e.g. `1970-01-01T10:00:00Z`.
`date_add(object, duration)`	adds the given ISO-8601 duration `duration` e.g. `P1D` or `-PT10H` to the date provided by the first parameter.
`to_string(object[, format[, timezone]])`	Converts given object into a string. If the format argument is present, it must be a Java Formatter [http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html] compliant string e.g. `%f` or `%tY-%tm-%td`. The timezone argument is significant if the object is a Date. If the timezone argument is specified it must be a valid Java timezone name. The date is converted to the specified timezone before being formatted by the`format`. If the timezone is omitted `UTC` is assumed.

6.3.12. Query Engine

6.3.12.1. Introduction

Broker query engine extends existing functionality of broker query API and allows executing complex SQL-like queries against the broker. It allows using predicates combining AND/OR/NOT logical operations, supports aggregation and grouping as well as numerous numeric, datetime and string functions. Currently, querying from multiple object types (domains) in a single query as well as all types of joins are not supported.

6.3.12.2. Broker Configuration

Some properties influencing the query output can be specified directly in the request, but there are also global properties, affecting the way query engine works.

Table 6.5. Query Engine Configuration

Context Property Name	Description
qpid.port.http.query.engine.cacheSize	Query cache size
qpid.port.http.query.engine.maxQueryDepth	Maximal query depth

Context Property Name	Description
qpid.port.http.query.engine.zoneId	Timezone ID

Query cache size

After query is parsed from the SQL string, it is stored into a cache. When the same query will be fired against the query engine, parsing will be omitted and the query structure will be retrieved from cache. By default, query cache size is 1000. This means, that when 1000 different queries will be fired against the query engine, the next one will override the oldest cache entry. When set to 0 or to negative value, query cache will not be used and each query will be parsed.

Maximal query depth

The longer is the query and the more conditions it contains, the bigger becomes the query depth. To limit query complexity, maximal query depth parameter can be used. By default, maximal query depth is 4096. This should suffice for most queries even complicated ones. If query depth exceeds this limit, following error will be returned:

```
{
    "errorMessage": "Max query depth reached: 4096"
}
```

Zone ID

Zone ID value should follow the rules described in javadoc [https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html#of-java.lang.String-]. The default value for zone id is "UTC".

6.3.12.3. Request Format

An authorized request should be sent to the following endpoint: POST http://<hostname>:<port>/api/latest/querybroker/broker SQL query should be supplied in the "sql" field of the JSON body:

```
{
    "sql": "select * from broker"
}
```

SQL Query Format

SQL keywords, operators and functions are case-insensitive, so are domain names (object types) specified in the `FROM` clause. Field names specified in the `SELECT` clause are case-sensitive. Following queries are similar:

```
{
    "sql": "SELECT name FROM BROKER"
}
```

```
{
    "sql": "SELECT name FROM broker"
}
```

```
{
    "sql": "select name from broker"
}
```

They will return the same output. When an entity field name is written in wrong case or misspelled, an error will be returned. For example, following query

```
{
    "sql": "SELECT NAME FROM BROKER"
}
```

has field `NAME` written in upper case, which will result in an error:

```
{
    "errorMessage": "Domain 'BROKER' does not contain field 'NAME'"
}
```

In this document many SQL queries are split into several lines for better readability, but JSON format does not support multiline string fields. Therefore, even the long SQL queries should be placed in `sql` field of the JSON body as a single line. Aside from SQL query several configuration parameters can be provided to influence output format:

Table 6.6. Additional Request Parameters

Field Name	Description
dateTimeFormat	Format of the datetime fields, possible values: LONG, STRING
dateTimePattern	Pattern for datetime fields formatting, e.g. yyyy-MM-dd HH:mm:ss
decimalDigits	Amount of decimal digits
roundingMode	Rounding mode for arithmetic operations, possible values UP, DOWN, CEILING, FLOOR, HALF_UP, HALF_DOWN, HALF_EVEN, UNNECESSARY

Datetime Format

When datetime format is specified as `LONG`, datetime fields will be returned as milliseconds from UNIX epoch. So, following query

```
{
```

```
    "sql": "select id, name, createdTime from broker",
    "dateTimeFormat": "LONG"
}
```

returns following result:

```
{
    "results": [
        {
            "id": "ce8bbaf0-3efa-4176-889a-7987ac1988cc",
            "name": "broker",
            "createdTime": 1645195849272
        }
    ],
    "total": 1
}
```

In opposite the query

```
{
    "sql": "select id, name, createdTime from broker",
    "dateTimeFormat": "STRING"
}
```

returns following result:

```
{
    "results": [
        {
            "id": "ce8bbaf0-3efa-4176-889a-7987ac1988cc",
            "name": "broker",
            "createdTime": "2022-02-18 15:50:49.272"
        }
    ],
    "total": 1
}
```

Datetime Pattern

The default format of the string datetime representation is "yyyy-MM-DD HH:mm:ss.SSS". It can be changed using the parameter `dateTimePattern`. The query

```
{
    "sql": "select id, name, createdTime from broker",
    "dateTimeFormat": "STRING",
    "dateTimePattern": "yyyy/MM/dd HH:mm:ss.SSS"
}
```

returns following result

```
{
    "results": [
        {
            "id": "ce8bbaf0-3efa-4176-889a-7987ac1988cc",
            "name": "broker",
            "createdTime": "2022/02/18 15:50:49.272"
        }
    ],
    "total": 1
}
```

Decimal Digits

By default, decimal digits value is 6, meaning there will be 6 digits after decimal point. For example, following query

```
{
    "sql": "select avg(queueDepthMessages) from queue"
}
```

returns following result:

```
{
    "results": [
        {
            "avg(queueDepthMessages)": 0.437227
        }
    ],
    "total": 1
}
```

This behavior can be changed for each value separately using ROUND or TRUNC functions, but can also be changed for the whole query result by supplying `decimalDigits` parameter. Following query

```
{
    "sql": "select avg(queueDepthMessages) from queue",
    "decimalDigits": 2
}
```

returns following result:

```
{
```

```
    "results": [
        {
            "avg(queueDepthMessages)": 0.43
        }
    ],
    "total": 1
}
```

Rounding Mode

Rounding mode affects how results of the arithmetic operations will be rounded. The rules of applying different rounding modes can be found in appropriate javadoc [https://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html]. Default rounding mode is HALF_UP. Changing rounding mode will affect division operations, but will not affect results of ROUND() and TRUNC() functions (which always use rounding mode HALF_UP and HALF_DOWN appropriately). Following query

```
{
    "sql": "select 2/3",
    "decimalDigits": 2,
    "roundingMode": "DOWN"
}
```

uses rounding mode `DOWN` and returns following result:

```
{
    "results": [
        {
            "2/3": 0.66
        }
    ],
    "total": 1
}
```

When rounding mode will be changed to `UP`

```
{
    "sql": "select 2/3",
    "decimalDigits": 2,
    "roundingMode": "UP"
}
```

result will be changed as well:

```
{
    "results": [
        {
            "2/3": 0.67
```

```
            }
        ],
        "total": 1
    }
```

6.3.12.4. Object Types (Domains)

Object types or domains to query from are specified in the "FROM" clause. The broker object hierarchy can be retrieved using an endpoint http://<hostname>:<port>/service/metadata Alternatively following SQL query can be fired

```
{
    "sql": "select * from domain"
}
```

returning similar result:

```
{
    "results": [
        {
            "name": "AccessControlProvider"
        },
        {
            "name": "AclRule"
        },
        {
            "name": "AuthenticationProvider"
        },
        {
            "name": "Binding"
        },
        {
            "name": "BrokerConnectionLimitProvider"
        },
        {
            "name": "BrokerLogInclusionRule"
        },
        {
            "name": "BrokerLogger"
        },
        {
            "name": "Certificate"
        },
        {
            "name": "Connection"
        },
        {
            "name": "ConnectionLimitRule"
        },
        {
```

```
    "name": "Consumer"
},
{
    "name": "Domain"
},
{
    "name": "Exchange"
},
{
    "name": "Group"
},
{
    "name": "GroupMember"
},
{
    "name": "GroupProvider"
},
{
    "name": "KeyStore"
},
{
    "name": "Plugin"
},
{
    "name": "Port"
},
{
    "name": "Queue"
},
{
    "name": "RemoteReplicationNode"
},
{
    "name": "Session"
},
{
    "name": "TrustStore"
},
{
    "name": "User"
},
{
    "name": "VirtualHost"
},
{
    "name": "VirtualHostAccessControlProvider"
},
{
    "name": "VirtualHostAlias"
},
{
    "name": "VirtualHostConnectionLimitProvider"
},
{
```

```
                "name": "VirtualHostLogInclusionRule"
            },
            {
                "name": "VirtualHostLogger"
            },
            {
                "name": "VirtualHostNode"
            }
        ],
        "total": 31
    }
```

In addition to the object types supported by broker query REST API, following object types (domains) can be used as well:

Table 6.7. Query Engine Domains

Domain
AclRule
Binding
Certificate
ConnectionLimitRule
Domain

Those objects do not belong to the broker object hierarchy (as they don't descend from ConfiguredObject), they were added to make queries against listed domains more simple. For example, following query

```
SELECT *
FROM AclRule
WHERE identity = 'amqp_user1'
```

returns following result:

```
{
    "results": [
        {
            "identity": "amqp_user1",
            "attributes": {},
            "action": {
                "objectType": "VIRTUALHOST",
                "properties": {
                    "name": null,
                    "empty": true
                },
                "operation": "ACCESS"
            },
            "objectType": null,
            "operation": null,
            "outcome": "ALLOW_LOG"
```

```
        },
        {
            "identity": "amqp_user1",
            "attributes": {
                "NAME": "request.amqp_user1",
                "ROUTING_KEY": "*"
                },
            "action": {
                "objectType": "EXCHANGE",
                "properties": {
                    "name": "request.amqp_user1",
                    "empty": false
                    },
                "operation": "PUBLISH"
            },
            "objectType": null,
            "operation": null,
            "outcome": "ALLOW"
        },
        {
            "identity": "amqp_user1",
            "attributes": {
                "NAME": "broadcast.amqp_user1.*"
                },
            "action": {
                "objectType": "QUEUE",
                "properties": {
                    "name": "broadcast.amqp_user1.*",
                    "empty": false
                    },
                "operation": "CONSUME"
            },
            "objectType": null,
            "operation": null,
            "outcome": "ALLOW_LOG"
        },
        {
            "identity": "amqp_user1",
            "attributes": {
                "NAME": "response.amqp_user1"
                },
            "action": {
                "objectType": "QUEUE",
                "properties": {
                    "name": "response.amqp_user1",
                    "empty": false
                    },
                "operation": "CONSUME"
            },
            "objectType": null,
            "operation": null,
            "outcome": "ALLOW_LOG"
        }
    ],
```

```
    "total": 4
}
```

Please note, that keyword `FROM` isn't mandatory, it is possible to execute queries without it, when the result shouldn't retrieve any data from broker. Few examples of such queries would be:

```
SELECT CURRENT_TIMESTAMP()
```

```
SELECT DATE(CURRENT_TIMESTAMP())
```

```
SELECT (2 + 10) / 3
```

```
SELECT 2 * 5 > 12
```

6.3.12.5. Filtering Results

Filtering is achieved by using different operators groups in a `WHERE` clause. Operators can be divided into comparison operators, conditional operators and logical operators.

Broker Data Types

Broker entities have fields belonging to different java types: primitives (boolean, int, long, double), strings, datetime Date [https://docs.oracle.com/javase/8/docs/api/java/util/Date.html], LocalDate [https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html], LocalDateTime [https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html], Instant [https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html]. Object IDs are usually of UUID [https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html] type. Many values are enums. When comparing field values, they follow some implicit casting rules: enums and UUIDs are cast to strings, datetime values are cast to Instant [https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html], numeric values are cast to BigDecimal [https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html]. When casting string value to date, by default is used pattern "uuuu-MM-dd". That allows to run following queries:

```
SELECT *
FROM certificate
WHERE DATE(validUntil) = '2020-12-31'
```

Here string value is implicitly cast to Instant and both value are compared as Instant instances. When casting string to datetime, by default is used pattern "uuuu-MM-dd HH:mm:ss" with optional 0-6 second fractions. That allows to run following queries:

```
SELECT *
FROM certificate
WHERE DATE(validUntil) > '2020-12-31 23:59:59.999'
```

Here string value is implicitly cast to Instant as well and both value are compared as Instant instances. It is important to compare values of the same type, otherwise an error may be returned or query may be evaluated erroneously. For example, following query

```
SELECT *
FROM queue
WHERE durable = 'true'
```

will return an empty result, because field `durable` is of boolean type and comparing boolean value with a string 'true' will always return false. The correct query should be

```
SELECT *
FROM queue
WHERE durable = true
```

Keyword DISTINCT

To remove duplicates from the results keyword "DISTINCT" can be used. For example, query

```
SELECT overflowPolicy
FROM queue
```

will return results for all queues, but query

```
SELECT DISTINCT overflowPolicy
FROM queue
```

will return only several values.

Comparison Operators

BETWEEN

Definition and Usage

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

Syntax

```
BETWEEN(expression1 AND expression2)
BETWEEN(expression1, expression2)
BETWEEN expression1 AND expression2
BETWEEN expression1, expression2
```

Parameter Values

Table 6.8. BETWEEN Parameters

Parameter	Description
expression1	Lower threshold
expression2	Higher threshold

Examples

Find names of the queues having depth in messages between 1000 and 2000

```
SELECT
    name
FROM queue
WHERE queueDepthMessages BETWEEN (1000, 2000)
```

Find certificates expiring between 2024-12-01 and 2024-12-31

```
SELECT *
FROM certificate
WHERE DATE(validUntil) BETWEEN ('2024-12-01' AND '2024-12-31')
```

EQUAL

Definition and Usage

Equal operator is designated using "=" character. It allows comparison of boolean, datetime, numeric and string values. Both compared values must have same type.

Syntax

```
expression1 = expression2
```

Parameter Values

Table 6.9. EQUAL Parameters

Parameter	Description
expression1	Expression to compare to
expression2	Expression to compare with

Examples

Find queue by name

```
SELECT *
FROM queue
WHERE name = 'broadcast.amqp_user1.Public'
```

GREATER THAN

Definition and Usage

Greater than operator is designated using ">" character. It allows comparison of datetime, numeric and string values. Both compared values must have same type.

Syntax

```
expression1 > expression2
```

Parameter Values

Table 6.10. GREATER THAN Parameters

Parameter	Description
expression1	Expression to compare to
expression2	Expression to compare with

Examples

Find queues having message depth greater than 1000

```
SELECT *
FROM queue
WHERE queueDepthMessages > 1000
```

GREATER THAN OR EQUAL

Definition and Usage

Greater than or equal operator is designated using ">=" characters. It allows comparison of datetime, numeric and string values. Both compared values must have same type.

Syntax

```
expression1 >= expression2
```

Parameter Values

Table 6.11. GREATER THAN OR EQUAL Parameters

Parameter	Description
expression1	Expression to compare to
expression2	Expression to compare with

Examples

Find queues having message depth greater than or equal to 1000

```
SELECT *
FROM queue
WHERE queueDepthMessages >= 1000
```

IN

Definition and Usage

The IN operator allows specifying multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions. Alternatively IN operator can be used with a subquery. When a subquery is used, it should return only one value, otherwise an error will be returned.

Syntax

```
expression IN (value_1, value_2, ..., value_n)
expression IN (SELECT value FROM domain)
```

Parameter Values

Table 6.12. IN Parameters

Parameter	Description
expression	Expression to compare to
value_1 - value_n	Values to compare with

Examples

Find bindings having destination queue belonging to the list

```
SELECT *
FROM binding
WHERE destination IN ('broadcast.amqp_user1.Service1', 'broadcast.amqp_user1.S
```

Find bindings having destination queue with message depth between 1000 and 2000

```
SELECT *
FROM binding
WHERE destination IN (SELECT name FROM queue WHERE queueDepthMessages BETWEEN
```

IS NULL

Definition and Usage

The IS NULL operator is used to compare ordinary values with NULL values.

Syntax

```
expression IS NULL
```

Parameter Values

Table 6.13. IS NULL Parameters

Parameter	Description
expression	Expression to compare to NULL

Examples

Find queues having NULL description

```
SELECT *
FROM queue
WHERE description IS NULL
```

LESS THAN

Definition and Usage

Less than operator is designated using "<" character. It allows comparison of datetime, numeric and string values. Both compared values must have same type.

Syntax

```
expression1 < expression2
```

Parameter Values

Table 6.14. LESS THAN Parameters

Parameter	Description
expression1	Expression to compare to
expression2	Expression to compare with

Examples

Find queues having message depth less than 1000

```
SELECT *
FROM queue
WHERE queueDepthMessages < 1000
```

LESS THAN OR EQUAL

Definition and Usage

Less than or equal operator is designated using "<=" characters. It allows comparison of datetime, numeric and string values. Both compared values must have same type.

Syntax

```
expression1 <= expression2
```

Parameter Values

Table 6.15. LESS THAN OR EQUAL Parameters

Parameter	Description
expression1	Expression to compare to
expression2	Expression to compare with

Examples

Find queues having message depth less than or equal to 1000

```
SELECT *
FROM queue
WHERE queueDepthMessages <= 1000
```

LIKE

Definition and Usage

The LIKE operator is used to search for a specified pattern in a string. There are two wildcards often used in conjunction with the LIKE operator: the percent sign "%" represents zero, one, or multiple characters; the question mark "?" represents one, single character.

Syntax

```
expression LIKE pattern
expression LIKE pattern ESCAPE escapeCharacter
expression LIKE (pattern)
expression LIKE (pattern ESCAPE escapeCharacter)
```

Parameter Values

Table 6.16. LIKE Parameters

Parameter	Description
expression	Expression to compare to
pattern	Pattern to compare against
escapeCharacter	Character used to escape percent sign or question mark

Examples

Find queues having name starting with a string "broadcast"

```
SELECT *
FROM queue
WHERE name LIKE 'broadcast%'
```

Find queues with name containing string "amqp_user1"

```
SELECT *
FROM queue
WHERE name LIKE '%amqp_user1%'
```

Conditional Operators

CASE

Definition and Usage

The CASE statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

Syntax

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END
```

Parameter Values

Table 6.17. CASE Parameters

Parameter	Description
condition1 - conditionN	Conditions to estimate
result1 - resultN	Results to return

Examples

Group queues into good (< 60% of max depth), bad (60% - 90% of max depth) and critical (> 90% of max depth), count number of queues in each group. Consider queues with unlimited depth being good.

```
SELECT
    COUNT(*),
    CASE
        WHEN maximumQueueDepthMessages != -1 AND maximumQueueDepthBytes != -1
            AND (queueDepthMessages > maximumQueueDepthMessages * 0.9 OR queue
        THEN 'critical'
        WHEN maximumQueueDepthMessages != -1 AND maximumQueueDepthBytes != -1
```

```
            AND queueDepthMessages BETWEEN (maximumQueueDepthMessages * 0.6 AN
            OR queueDepthBytes BETWEEN (maximumQueueDepthBytes * 0.6 AND maxim
        THEN 'bad'
        ELSE 'good'
    END AS queueState
FROM queue
GROUP BY queueState
```

The "AND" and "OR" operators are used to filter records based on more than one condition: the "AND" operator displays a record if all the conditions separated by "AND" are TRUE. The "OR" operator displays a record if any of the conditions separated by "OR" is TRUE. The "NOT" operator displays a record if the condition(s) is NOT TRUE.

6.3.12.6. Sorting Results

Default sorting order is ascending, default sorting field is `name` for domains having this field. Results of the following query will be sorted ascending by name:

```
SELECT *
FROM queue
```

Few exceptions are following:

Table 6.18. Default sorting fields

Domain	Default sorting field
AclRule	identity
Certificate	alias
ConnectionLimitRule	identity

Results of the following query will be sorted ascending by alias:

```
SELECT *
FROM certificate
```

To apply another sorting rules clause `ORDER BY` should be used. It may contain one of the fields specified in the `SELECT` clause:

```
SELECT
    id, name, state
FROM queue
ORDER BY name
```

Alternatively it may contain fields not specified in `SELECT` clause:

```
SELECT
```

```
    id, name, state
FROM queue
ORDER BY overflowPolicy
```

Instead of using field names or aliases items in the `ORDER BY` clause can also be referenced by ordinal - the numeric value of their order of appearance in the `SELECT` clause. For example, following query

```
SELECT
    name, overflowPolicy
FROM queue
ORDER BY 2 DESC, 1 ASC
```

will return results sorted in descending order by overflow policy and inside the groups with the same overflow policy name results will be sorted by queue name in ascending order.

6.3.12.7. Aggregation

Aggregation is achieved using functions AVG(), COUNT(), MAX(), MIN() and SUM(). It's important to remember, that aggregation functions don't consider NULL values. For example, following query

```
SELECT COUNT(description)
FROM queue
```

will return count of queues having non-null value of a field `description`. To consider NULL values, they should be handled using COALESCE() function or CASE operator:

```
SELECT COUNT(COALESCE(description, ''))
FROM queue
```

Alternatively

```
SELECT COUNT(CASE WHEN description IS NULL THEN '' ELSE description END)
FROM queue
```

Several aggregation functions can be used together in the same query:

```
SELECT
    COUNT(*),
    AVG(queueDepthMessages),
    SUM(queueDepthMessages),
    SUM(queueDepthBytes),
    MIN(queueDepthMessages),
    MAX(queueDepthMessages),
    MIN(queueDepthBytes),
    MAX(queueDepthBytes)
FROM queue
```

6.3.12.8. Grouping

Grouping of the aggregated results can be achieved using the `GROUP BY` clause. For example, following query finds count of ACL rules for each user and output them in descending order:

```
SELECT
    COUNT(*) AS cnt, identity
FROM aclrule
GROUP BY identity
ORDER BY 1 DESC
```

The result of the query:

```
{
    "results": [
        {
            "cnt": {
                "amqp_user1": 6,
                "amqp_user2": 4,
                "amqp_user3": 4,
                ... some results ommited ...
                "amqp_user97": 2,
                "amqp_user98": 1,
                "amqp_user99": 1
            }
        }
    ],
    "total": 1
}
```

To filter the grouped result `HAVING` clause can be used:

```
SELECT
    overflowPolicy, COUNT(*)
FROM queue
GROUP BY overflowPolicy
HAVING SUM(queueDepthMessages) > 1000
```

6.3.12.9. Functions

Aggregation Functions

AVG

Definition and Usage

The AVG() function returns the average value of a collection.

Syntax

```
AVG(expression)
```

Parameter Values

Table 6.19. AVG Parameters

Parameter	Description
expression	Expression result average value of which should be found

Examples

Find average amount of bytes used by queues with names starting with "broadcast"

```
SELECT
    AVG(queueDepthBytes)
FROM queue
WHERE name LIKE 'broadcast%'
```

COUNT

Definition and Usage

The COUNT() function returns the number of items that matches a specified criterion.

Syntax

```
COUNT(expression)
COUNT(DISTINCT expression)
```

Parameter Values

Table 6.20. COUNT Parameters

Parameter	Description
expression	Expression result of which should be counted

Examples

Find amount of queues with names starting with "broadcast"

```
SELECT
    COUNT (*)
FROM queue
WHERE name LIKE 'broadcast%'
```

MAX

Definition and Usage

The MAX() function returns the maximum value of a collection.

Syntax

```
MAX(expression)
```

Parameter Values

Table 6.21. MAX Parameters

Parameter	Description
expression	Expression result maximal value of which should be found

Examples

Find maximal amount of bytes used by queues with names starting with "broadcast"

```
SELECT
    MAX(queueDepthBytes)
FROM queue
WHERE name LIKE 'broadcast%'
```

MIN

Definition and Usage

The MIN() function returns the minimum value of a collection.

Syntax

```
MIN(expression)
```

Parameter Values

Table 6.22. MIN Parameters

Parameter	Description
expression	Expression result minimal value of which should be found

Examples

Find minimal amount of bytes used by queues with names starting with "broadcast"

```
SELECT
    MIN(queueDepthBytes)
FROM queue
WHERE name LIKE 'broadcast%'
```

SUM

Definition and Usage

The SUM() function returns the total sum of a numeric collection.

Syntax

```
SUM(expression)
```

Parameter Values

Table 6.23. SUM Parameters

Parameter	Description
expression	Expression result of which should be summed

Examples

Find amount of bytes used by queues having names starting with "broadcast"

```
SELECT
    SUM(queueDepthBytes)
FROM queue
WHERE name LIKE 'broadcast%'
```

Datetime Functions

CURRENT_TIMESTAMP

Definition and Usage

The CURRENT_TIMESTAMP() function returns current date and time.

Syntax

```
CURRENT_TIMESTAMP()
```

Parameter Values

Function has no parameters

Examples

Find current date and time

```
SELECT CURRENT_TIMESTAMP()
```

DATE

Definition and Usage

The DATE() function extracts the date part from a datetime expression.

Syntax

```
DATE(expression)
```

Parameter Values

Table 6.24. DATE Parameters

Parameter	Description
expression	A valid date/datetime value

Examples

Find certificates having validFrom equal to 01. January 2020

```
SELECT *
FROM certificate
WHERE DATE(validFrom) = '2020-01-01'
```

Find certificates expiring between 01. January 2020 and 10. January 2020

```
SELECT *
FROM certificate
WHERE DATE(validUntil) BETWEEN ('2020-01-01', '2020-01-10')
```

DATEADD

Definition and Usage

The DATEADD() function adds a time/date interval to a date and returns the date.

Syntax

```
DATEADD(TIMEUNIT, VALUE, DATE)
```

Parameter Values

Table 6.25. DATEADD Parameters

Parameter	Description
TIMEUNIT	The type of time unit to add. Can be one of the following values: YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND
VALUE	The value of the time/date interval to add. Both positive and negative values are allowed
DATE	The date to be modified

Examples

Find certificates expiring in less than 30 days

```
SELECT *
FROM certificate
WHERE DATEADD(DAY, -30, validUntil) < CURRENT_TIMESTAMP()
LIMIT 10 OFFSET 0
```

DATEDIFF

Definition and Usage

The DATEDIFF() function returns the number of time units between two date values.

Syntax

```
DATEDIFF(TIMEUNIT, DATE1, DATE2)
```

Parameter Values

Table 6.26. DATEDIFF Parameters

Parameter	Description
TIMEUNIT	Time unit to calculate difference. Can be one of the following values: YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND
DATE1	Start date
DATE2	End date

Examples

Find certificate aliases and days until expiry

```
SELECT
    alias,
    DATEDIFF(DAY, CURRENT_TIMESTAMP(), validUntil) AS days_until_expiry
FROM certificate
LIMIT 10 OFFSET 0
```

EXTRACT

Definition and Usage

The EXTRACT() function extracts a part from a given date.

Syntax

```
EXTRACT(TIMEUNIT FROM DATE)
```

Parameter Values

Table 6.27. EXTRACT Parameters

Parameter	Description
TIMEUNIT	Time unit to extract. Can be one of the following values: YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND
DATE	The date to extract a part from

Examples

Find certificates issued in January 2020

```
SELECT *
FROM certificate
WHERE EXTRACT(YEAR FROM validFrom) = 2020
AND EXTRACT(MONTH FROM validFrom) = 1
LIMIT 10 OFFSET 0
```

NULL Functions

COALESCE

Definition and Usage

The COALESCE() function returns the first non-null value in a list.

Syntax

```
COALESCE(value_1, value_2, ...., value_n)
```

Parameter Values

Table 6.28. COALESCE Parameters

Parameter	Description
value_1 - value_n	The values to test

Examples

Find count of queues having NULL description

```
SELECT
    COUNT(COALESCE(description, 'empty')) AS RESULT
FROM queue
HAVING COALESCE(description, 'empty') = 'empty'
```

Numeric Functions

ABS

Definition and Usage

The ABS() function returns the absolute value of a number.

Syntax

```
ABS(number)
```

Parameter Values

Table 6.29. ABS Parameters

Parameter	Description
number	A numeric value

Examples

Find absolute amount of days after the validFrom date of the certificates

```
SELECT
    ABS(DATEDIFF(DAY, CURRENT_TIMESTAMP(), validFrom))
FROM certificate
```

ROUND

Definition and Usage

The ROUND() function takes a numeric parameter and rounds it to the specified number of decimal places.

Syntax

```
ROUND(number, decimals)
```

Parameter Values

Table 6.30. ROUND Parameters

Parameter	Description
number	The number to be rounded
decimals	The number of decimal places to round to

Examples

Find average queue depth in messages and round result to 2 decimal places

```
SELECT
    ROUND(AVG(queueDepthMessages)) as result
FROM queue
```

TRUNC

Definition and Usage

The TRUNC() function takes a numeric parameter and truncates it to the specified number of decimal places.

Syntax

```
TRUNC(number, decimals)
```

Parameter Values

Table 6.31. TRUNC Parameters

Parameter	Description
number	The number to be truncated
decimals	The number of decimal places to truncate to

Examples

Find average queue depth in messages and truncate result to 2 decimal places

```
SELECT
    TRUNC(AVG(queueDepthMessages)) as result
FROM queue
```

String Functions

CONCAT

Definition and Usage

The CONCAT() function takes a variable number of arguments and concatenates them into a single string. It requires a minimum of one input value, otherwise CONCAT will raise an error. CONCAT implicitly converts all arguments to string types before concatenation. The implicit conversion to strings follows the existing rules for data type conversions. If any argument is NULL, CONCAT returns NULL.

Syntax

```
CONCAT(expression_1, expression_2, expression_3, ..., expression_n)
```

Parameter Values

Table 6.32. LENGTH Parameters

Parameter	Description
expression_1 - expression_n	The expressions to add together

Examples

Output certificate alias and validity dates using format "alias: validFrom - validUntil"

```
SELECT
    CONCAT(alias, ': ', DATE(validFrom), ' - ', DATE(validUntil)) as validity
FROM certificate
```

LEN / LENGTH

Definition and Usage

The LEN() / LENGTH() function takes a string parameter and returns its length. The implicit conversion to strings follows the existing rules for data type conversions. If any argument is NULL, LEN / LENGTH returns 0.

Syntax

```
LEN(string)
LENGTH(string)
```

Parameter Values

Table 6.33. LENGTH Parameters

Parameter	Description
string	The string to count the length for

Examples

Find certificate aliases having alias length greater than 10

```
SELECT
    alias
FROM certificate
WHERE LENGTH(alias) > 10
LIMIT 10 OFFSET 0
```

LOWER

Definition and Usage

The LOWER() function takes a string parameter and converts it to lower case. The implicit conversion to strings follows the existing rules for data type conversions. If argument is NULL, LOWER returns NULL.

Syntax

```
LOWER(string)
```

Parameter Values

Table 6.34. LOWER Parameters

Parameter	Description
string	The string to convert

Examples

Filter connections by principal name (case-insensitive)

```
SELECT *
FROM connection
WHERE LOWER(principal) = 'amqp_user1'
LIMIT 10 OFFSET 0
```

LTRIM

Definition and Usage

The LTRIM() function removes leading spaces from a string. If argument is NULL, RTRIM returns NULL.

Syntax

```
LTRIM(string)
LTRIM(string, chars)
```

Parameter Values

Table 6.35. LTRIM Parameters

Parameter	Description
string	The string to remove leading and trailing spaces from
chars	Specific characters to remove

Examples

Find connection remote addresses

```
SELECT
    LTRIM(remoteAddress, '/') AS remoteAddress
FROM connection
```

POSITION

Definition and Usage

The POSITION() function takes a search pattern and a source string as parameters and returns the position of the first occurrence of a pattern in a source string. If the pattern is not found within the source string, this function returns 0. Optionally takes third integer parameter, defining from which position search should be started. Third parameter should be an integer greater than 0. If source string is NULL, returns zero.

Syntax

```
POSITION(pattern IN source)
POSITION(pattern IN source, startIndex)
```

Parameter Values

Table 6.36. POSITION Parameters

Parameter	Description
pattern	The pattern to search for in source
source	The original string that will be searched
startIndex	The index from which search will be started

Examples

Find queues having string "broadcast" in their names

```
SELECT *
FROM queue
WHERE POSITION('broadcast', name) > 0
LIMIT 10 OFFSET 0
```

REPLACE

Definition and Usage

The REPLACE() function replaces all occurrences of a substring within a string, with a new substring. If source string is NULL, returns NULL.

Syntax

```
REPLACE(source, pattern, replacement)
```

Parameter Values

Table 6.37. REPLACE Parameters

Parameter	Description
source	The original string
pattern	The substring to be replaced
replacement	The new replacement substring

Examples

Output certificate issuer names without leading "CN="

```
SELECT
    REPLACE(issuerName, 'CN=', '') AS issuer
FROM certificate
LIMIT 10 OFFSET 0
```

RTRIM

Definition and Usage

The RTRIM() function removes trailing spaces from a string. If argument is NULL, RTRIM returns NULL.

Syntax

```
RTRIM(string)
RTRIM(string, chars)
```

Parameter Values

Table 6.38. RTRIM Parameters

Parameter	Description
string	The string to remove leading and trailing spaces from
chars	Specific characters to remove

Examples

Find connection remote addresses

```
SELECT
    RTRIM(remoteAddress)
FROM connection
```

SUBSTR / SUBSTRING

Definition and Usage

The SUBSTRING() function takes a source parameter, a start index parameter and optional length parameter. Returns substring of a source string from the start index to the end or using the length parameter. If source string is NULL, return NULL.

Syntax

```
SUBSTRING(source, startIndex, length)
```

Parameter Values

Table 6.39. SUBSTRING Parameters

Parameter	Description
source	The string to extract from
startIndex	The start position. Can be both a positive or negative number. If it is a positive number, this function extracts from the beginning of the string. If it is a negative number, function extracts from the end of the string
length	The number of characters to extract. If omitted, the whole string will be returned (from the start position). If zero or negative, an empty string is returned

Examples

Find queue names removing from name part before the `.` character

```
SELECT
    SUBSTRING(name, POSITION('.', name) + 1, LEN(name) - POSITION('.', name))
FROM queue
```

TRIM

Definition and Usage

The TRIM() function removes both leading and trailing spaces from a string. If argument is NULL, TRIM returns NULL.

Syntax

```
TRIM(string)
TRIM(string, chars)
```

Parameter Values

Table 6.40. TRIM Parameters

Parameter	Description
string	The string to remove leading and trailing spaces from
chars	Specific characters to remove

Examples

Find connections remote addresses removing `/` characters from both sides

```
SELECT
    TRIM(remoteAddress, '/')
FROM connection
```

UPPER

Definition and Usage

The UPPER() function takes a string parameter and converts it to upper case. The implicit conversion to strings follows the existing rules for data type conversions. If argument is NULL, UPPER returns NULL.

Syntax

```
UPPER(string)
```

Parameter Values

Table 6.41. UPPER Parameters

Parameter	Description
string	The string to convert

Examples

Filter connections by principal name (case-insensitive)

```
SELECT *
FROM connection
WHERE UPPER(principal) = 'AMQP_USER1'
LIMIT 10 OFFSET 0
```

6.3.12.10. Set Operations

UNION, MINUS and INTERSECT set operations are supported. The UNION operator is used to combine the result-set of two or more SELECT statements. Every SELECT statement within UNION must have the same number of columns. The UNION operator selects distinct values by default. To keep duplicates, UNION ALL should be used. For example, following query return certificate aliases along with the user names:

```
SELECT UPPER(alias)
FROM certificate
UNION
SELECT UPPER(name)
FROM user
```

The MINUS operator is used to remove the results of right SELECT statement from the results of left SELECT statement. Every SELECT statement within MINUS must have the same number of columns. The MINUS operator selects distinct values by default. To eliminate duplicates, MINUS ALL should be used. For example, following query finds queue names, not specified as binding destinations:

```
SELECT name
FROM queue
MINUS
SELECT destination
FROM binding
```

The INTERSECT operation is used to retain the results of right SELECT statement present in the results of left SELECT statement. Every SELECT statement within INTERSECT must have the same number of columns. The INTERSECT operator selects distinct values by default. to eliminate duplicates, INTERSECT ALL should be used. For example, following query finds certificate aliases similar with the user names:

```
SELECT UPPER(alias)
FROM certificate
INTERSECT
SELECT UPPER(name)
FROM user
```

6.3.12.11. Subqueries

When executing subquery parent query domain mat be passed into the subquery using alias. E.g. this query

```
SELECT
    id,
    name,
    (SELECT name FROM connection WHERE SUBSTRING(name, 1, POSITION(']' IN name
    (SELECT id FROM connection WHERE SUBSTRING(name, 1, POSITION(']' IN name))
```

```
    (SELECT name FROM session WHERE id = c.session.id) as session
FROM consumer c
```

returns following result:

```
{
    "results": [
        {
            "id": "7a4d7a86-652b-4112-b535-61272b936b57",
            "name": "1|1|qpid-jms:receiver:ID:6bd18833-3c96-4936-b9ee-9dec5f40
            "connection": "[1] 127.0.0.1:39134",
            "connectionId": "afbd0480-43b1-4b39-bc00-260c077095f3",
            "session": "1"
        }
    ],
    "total": 1
}
```

Query

```
SELECT
    name,
    destination,
    (SELECT id FROM queue WHERE name = b.destination) AS destinationId,
    exchange,
    (SELECT id FROM exchange WHERE name = b.exchange) AS exchangeId
FROM binding b
WHERE name = 'broadcast.amqp_user1.xxx.#'
```

returns following result:

```
{
    "results": [
        {
            "name": "broadcast.amqp_user1.xxx.#",
            "destination": "broadcast.amqp_user1.xxx",
            "destinationId": "d5ce9e78-8558-40db-8690-15abf69ab255",
            "exchange": "broadcast",
            "exchangeId": "470273aa-7243-4cb7-80ec-13e698c36158"
        },
        {
            "name": "broadcast.amqp_user1.xxx.#",
            "destination": "broadcast.amqp_user2.xxx",
            "destinationId": "88357d15-a590-4ccf-aee8-2d5cda77752e",
            "exchange": "broadcast",
            "exchangeId": "470273aa-7243-4cb7-80ec-13e698c36158"
        },
        {
            "name": "broadcast.amqp_user1.xxx.#",
```

```
                "destination": "broadcast.amqp_user3.xxx",
                "destinationId": "c8200f89-2587-4b0c-a8f6-120cda975d03",
                "exchange": "broadcast",
                "exchangeId": "470273aa-7243-4cb7-80ec-13e698c36158"
            }
        ],
        "total": 3
    }
```

Query

```
    SELECT
        alias,
        (SELECT COUNT(id) FROM queue WHERE POSITION(UPPER(c.alias) IN name) > 0) A
    FROM certificate c
```

returns following result:

```
    {
        "results": [
            {
                "alias": "xxx",
                "queueCount": 5
            },
            {
                "alias": "xxy",
                "queueCount": 5
            },
            {
                "alias": "xxz",
                "queueCount": 7
            }
        ],
        "total": 3
    }
```

6.3.12.12. Performance Tips

Try to select entity fields by names instead of using an asterix. For example, this query

```
    SELECT
        id, name, state, overflowPolicy, expiryPolicy
    FROM queue
```

will be executed faster than this one:

```
    SELECT *
```

```
FROM queue
```

Try to use `LIMIT` and `OFFSET` clauses where applicable to reduce the response JSON size:

```
SELECT
    id, name, state, overflowPolicy, expiryPolicy
FROM queue
LIMIT 10 OFFSET 0
```

6.3.13. Cross Origin Resource Sharing (CORS)

The Broker supports Cross Origin Resource Sharing (CORS) to allow web management consoles other than the one embedded in the broker to use the REST API. This feature must be enabled by configuring the CORS Allow Origins and related attributes on the Section 7.17, “HTTP Plugin”

6.4. Prometheus Metrics

This section describes the metrics endpoints exposing broker statistics in Prometheus format [https://prometheus.io/]. The metrics endpoint is intended for scraping by Prometheus server to collect the Broker telemetry.

The Prometheus metric endpoints are mapped under /metrics path and /metrics/*. The latter allows to get the Virtual Host statistics by specify the path to the virtual host as /metrics/<virtual host node name>/<virtual host name>. The former allow to get all Broker statistics or Virtual Host statistics when called with HOST header set to the Virtual Host name

The metrics endpoints allow anonymous access by default. If required, an authentication can be enabled for the metrics endpoints by setting http management context variable `qpid.httpManagement.enableMetricContentAuthentication` to `true`.

The Broker JVM statistics are disabled by default. The metrics endpoints can be called with parameter `includeDisabled` set to `true` to include JVM broker metrics into endpoint output. If required, the JVM metrics could be enabled by setting context variable `qpid.metrics.includeDisabled` to `true`.

Note

For more information about Prometheus, check out the prometheus documentation [https://prometheus.io/docs/introduction/overview/].

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.

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. 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.

7.2.1. Attributes

- *Name the Broker*. This helps distinguish between Brokers in environments that have many.

- *Shutdown timeout*. Broker shutdown timeout in seconds (disabled if 0). If clean shutdown takes more than shutdown timeout, broker exits immediately.
- *Confidential configuration encryption provider*. The name of the provider used to encrypt passwords and other secrets within the configuration. See Section 8.5, “Configuration Encryption”.

7.2.2. Context

- *broker.flowToDiskThreshold* Controls the flow to disk feature.
- *broker.messageCompressionEnabled* Controls the message compression .
- *store.filesystem.maxUsagePercent* Maximum percentage of space that may be utilised on a filesystem hosting a virtualhost's message store before producer flow control is automatically imposed.

  This defaults to 90%.
- *qpid.broker_default_supported_protocol_version_reply* 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_msg_auth* 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 value can be overridden for each Virtual Host by setting the context value on the Virtual Host or Virtual Host Node.

  Defaults to false.

7.2.3. Children

- Virtualhost nodes
- Ports
- Authentication Providers
- Key Stores / Trust Stores
- Group Providers
- Access Control Providers
- Connection Limit Providers

7.2.4. Lifecycle

Not supported

7.3. Virtualhost Nodes

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

7.3.1. Types

The following virtualhost nodes types are supported.

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

7.3.2. 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.
  - 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”
- *Allow this node to operate solo* (HA only - groups of two). Allows a single node in a two node group to operate solo. See Section 10.4.5, “Allow to Operate Solo”

7.3.3. Children

- Virtualhost
- Remote Replication Nodes

[1] JDBC 4.0 compatible drivers must be available. See Section F.2, “Installing External JDBC Driver”

7.3.4. 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.

7.4. 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.

7.4.1. Types

The following virtualhost types are supported.

- BDB - Virtualhost backed with Oracle Berkeley DB JE
- BDB HA - Virtualhost backed with Oracle BDB utilising High Availability
- DERBY - Virtualhost backed with Apache Derby
- JDBC - Virtualhost backed with an external database [2]
- Memory - In-memory node (changes lost on Broker restart)
- Provided - Virtualhost that co-locates message data within the parent virtualhost node [3].

7.4.2. Context

- *use_async_message_store_recovery* Controls the background recovery feature.

7.4.3. Attributes

- *Name the virtualhost*. This is the name the messaging clients refer to when forming a connection to the Broker.

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

- *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”.
- *Connection thread pool size*. Number of worker threads used to perform messaging with connected clients.

  Defaults to 64 or double the maximum number of available processors, whichever is the larger.
- *Number of selectors*. Number of worker threads used from the thread pool to dispatch I/O activity to the worker threads.

  Defaults to one eighth of the thread pool size. Minimum 1.
- *Store transaction timeouts*. Warns of long running producer transactions. See Section 9.3, “Transaction Timeout”
- *Synchronization policy*. HA only. See Section 10.4.2, “Synchronization Policy”

7.4.4. Children

- Exchange
- Queue

7.4.5. 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.

7.5. 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.

7.5.1. 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.

7.5.2. Children

None

7.5.3. 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.

7.5.4. 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.

7.6. Exchanges

Exchanges can be managed using the HTTP or AMQP channels.

7.6.1. Types

- Direct
- Topic
- Fanout
- Headers

7.6.2. 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.

- *Durable*. Whether the exchange survives a restart.
- *alternateBinding*. Provides an alternate destination that, depending on behaviour requested by the producer, may be used if a message arriving at this exchange cannot be routed to at least one queue.
- *unroutableMessageBehaviour*. (AMQP 1.0 only) Default behaviour to apply when a message is not routed to any queues.

7.6.3. Lifecycle

Not supported

7.7. Queues

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

Queues can be managed using the HTTP or AMQP channels.

7.7.1. 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

7.7.2. 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 (TTL). Messages arriving with TTL larger than the maximum (including those with no TTL at all, which are considered to have a TTL of infinity) will be overridden by the maximum. Similarly, messages arriving with TTL less than the minimum, 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.
- *Overflow policy*. 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 4.7.5, “Controlling Queue Size”.

- *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 E, *Queue Alerts*
- *Message Groups*. See Section 4.7.2, “Messaging Grouping”
- *maximumDeliveryAttempts*. See Section 9.4, “Handing Undeliverable Messages”
- *alternateBinding*. Provides an alternate destination that will be used when the number of delivery attempts exceeds the *maximumDeliveryAttempts* configured on this queue. Messages are also routed to this destination if this queue is deleted.

7.7.3. Lifecycle

Not supported

7.8. 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*.

7.8.1. 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.

7.9. Producers

A Producers represents an application sending messages to an exchange or a queue. Its presence in the model indicates that an application is currently connected to the exchange or to the queue *at this moment*. Producers are created when using AMQP 1.0 protocol.

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 for management.

A port is defined to have one or more transports. A transport can either be plain (TCP) or SSL (TLS). 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.

Ports can only be managed by the HTTP management channel.

7.10.1. Context

- *qpid.port.max_open_connections*. The default maximum number of concurrent connections supported by an AMQP port.
- *qpid.port.amqp.acceptBacklog*. The backlog is the maximum number of pending connections that may be queued by the AMQP port. Once the queue is full, further connections will be refused. This is a

request to the operating system which may or may not be respected. The operating system itself may impose a ceiling. [4]

- *qpid.port.heartbeatDelay*. For AMQP 0-8..0-10 the default period with which Broker and client will exchange heartbeat messages (in seconds). Clients may negotiate a different heartbeat frequency or disable it altogether. For AMQP 1.0 this setting controls the incoming idle timeout only. A value of 0 disables.

7.10.2. 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.
- *Transports*. A list of transports supported by the port. For messaging or HTTP management chose TCP, SSL or both.
- *Enabled/Disabled Cipher Suites*. Allows cipher suites supported by the JVM to be enabled or disabled. The cipher suite names are those understood by the JVM.

  SSLv3 is disabled by default.
- *Keystore*. Keystore containing the Broker's private key. Required if 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.
- *Truststore*. Trust store contain an issuer certificate or the public keys of the clients themselves if peers only is desired.
- *Maximum Open Connections*. AMQP ports only. Limits the number of connections that may be open at any one time.
- *Thread pool size*. AMQP ports only. Number of worker threads used to process AMQP connections during connection negotiation phase.

  Defaults to 8.
- *Number of selectors*. AMQP ports only. Number of worker threads used from the thread pool to dispatch I/O activity to the worker threads.

  Defaults to one eighth of the thread pool size. Minimum 1.

7.10.3. Children

- Connection

[4] Some Linux distributions govern the ceiling with a `sysctl` setting `net.core.somaxconn`.

7.10.4. 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.
- OAuth2: authenticate users against a OAuth2 Authorization 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 for messaging and/or management.

The Broker supports a number of different keystore types. These are described below.

The key material may be held by the Broker itself (held inline within the configuration) or you may use references to files on the server's file system. Whichever mechanism is chosen it is imperative to ensure that private key material remains confidential.

7.12.1. Types

The following keystore types are supported.

- *File Key Store*. This type accepts the standard JKS keystore format undertood by Java and Java tools such as keytool [http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html].

  If the keystore contains multiple keys, it is possible to indicate which certificate is to be used by specifying an alias. If no alias is specified the first certificate found in the keystore will be used.

- *Non Java Key Store*. A Non Java Keystore accepts key material in PEM and DER file formats. With this store type it is necessary to provide the private key, which must not be protected by password, certificate and optionally a file containing intermediate certificates.

- *Auto Generated Self Signed* has the ability to generate a self signed certificate and produce a truststore suitable for use by an application using the Apache Qpid JMS and Apache Qpid JMS AMQP 0-x clients.

  The use of self signed certficates is not recommended for production use.

7.12.2. Attributes

- *Name the keystore*. Used to identify the keystore.

The following attributes apply to *File Key Stores* only.

- *Keystore path*. File Key Stores only. 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 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.
- *Use SNI host name matching*. If selected, SNI server name from an SSL handshake will be used to select the most appropriate certificate by matching an indicated hostname with the certificate hostname specified in subject or subject alternatives as CN or DC.

The following attributes apply to *Non Java Key Stores* only.

- *Private Key*. The private key in DER or PEM format. This file must not be password protected.
- *Certificate*. The cerificate in DER or PEM format.
- *Intermediates Certificates* . Optional. Intermediate cerificates in PEM or DER format.

The following attributes apply to *Auto Generated Self Signed* only.

- *Algorithm*. Optional. Algorithm used to generate the self-signed certificate.
- *Signature Algorithm* . Optional. The name of signature algorithm.
- *Key Length*. Optional. Length of the key in bits.
- *Duration*. Optional. Validility period in months.

7.12.3. Children

None

7.12.4. Lifecycle

Not supported

7.13. Truststores

Truststores have a number of roles within the Broker.

- A truststore is required by a Port in order to support SSL client authentication.
- Truststores have a optional role in end to end message encryption. The Broker acts as a Key Server [https://en.wikipedia.org/wiki/Key_server_(cryptographic)] so that publishing applications have convenient access to recipient's public keys.
- Some authentication providers also use a truststore when connecting to authentication systems that are protected by a private issuer SSL certificate.

7.13.1. Types

The following truststore types are supported.

- *File Trust Store*. This type accepts the standard JKS truststore format understood by Java and Java tools such as keytool [http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html].

- *Non Java Trust Store*. A non java trust store accepts key material in PEM and DER file formats. Either a path to the certificate on the server can be specified using the file:// protocol or the certificate can be uploaded with the data:// protocol
- *Managed Certificate Store*. This type accepts key material in PEM and DER file formats. Contrary to the Non Java Trust Store this store allows the user to add multiple certificates and stores them in the broker configuration.
- *Site Specific Trust Store*. This type will download a certificate from the provided SSL/TLS enabled URL. Note that you must specify both the protocol and the port. Example: https://example.com:443

7.13.2. Attributes

- *Name the truststore*. Used to identify the truststore.
- *Exposed as Message Source*. If enabled, the Broker will distribute certificates contained within the truststore to clients. Used by the end to end message encryption feature.
- *Trust Anchor Validity Enforced*. If enabled, authentications will fail if the trust anchor's validity date has not yet been reached or already expired.

Revocation attributes.

- *Enabled*. If set to true certificate revocation check is performed when client tries to connect.
- *Only End Entity*. If enabled, check only the revocation status of end-entity certificates.
- *Prefer CRLs*. If enabled, prefer CRL (specified in certificate distribution points) to OCSP, if disabled prefer OCSP to CRL.
- *No Fallback*. If enabled, disable fallback to CRL/OCSP (if *Prefer CRLs* set to true, disable fallback to OCSP, otherwise disable fallback to CRL in certificate distribution points).
- *Ignore Soft Failures*. If enabled, revocation check will succeed if CRL/OCSP response cannot be obtained because of network error or OCSP responder returns internalError or tryLater.
- *Server CRL Path Or Upload*. Path to Certificate Revocation List file. If set, certificate revocation check uses only set CRL file and ignores CRL Distribution Points in certificate.

The following attributes apply to *File Trust Stores* only.

- *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.

The following attributes apply to *Non Java Trust Stores* only.

- *Certificates*. The cerificate(s) in DER or PEM format.

7.13.3. Children

None

7.13.4. 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 Providers”

7.16. Connection Limit Providers

An Connection Limit Provider governs how many connections could user open on AMQP ports.

See Section 8.4, “Connection Limit Providers”

7.17. HTTP Plugin

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

7.17.1. 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.
- *CORS Allow Origins* is a comma separated list of origins that are allowed to access the REST API. Set to '*' to allow all origins. Default is empty, meaning CORS is disabled.

- *CORS Allow Methods* is a comma separated list of HTTP methods that are allowed to be used when doing CORS requests. Default value is "HEAD,GET,POST".
- *CORS Allow Headers* is a comma separated list of HTTP headers that are allowed to be specified when doing CORS requests. Default value is "Content-Type,Accept,Origin,X-Requested-With".
- *CORS Allow Credentials* is a boolean indicating if the resource allows requests with credentials. Default value is true.

7.17.2. Children

None

7.17.3. Lifecycle

Not supported

Important

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

Chapter 8. Security

8.1. Authentication Providers

In order to successfully establish a connection to the Broker, the connection must be authenticated. The 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).

- *Authentication method* is a method of authentication to use on binding into LDAP when `bind without search` mode is not selected. Supported methods are NONE, SIMPLE, GSSAPI. The latter requires setting of *Login Config Scope* which is a name of JAAS login module from JASS login configuration file specified using JVM system property *java.security.auth.login.config* or Java security properties file. If *Login Config Scope* is not specified with `GSSAPI` *Authentication method*, the scope *qpid-broker-j* will be used.

- *useFullLDAPName* is a boolean flag, defining whether user principal name will be obtained from full LDAP DN (default behavior) or from CN only. Set this flag to "false" to be able referencing principal by its CN in broker ACL rules.

- Additional group information can be obtained from LDAP. There are two common ways of representing group membership in LDAP.

  - User entries can hold membership information as attribute. To use this the *attribute name* that holds the group information must be specified.

  - Group entries can hold a list of their members as attribute. This can be used by specifying a *search context* and *search filter* to find all groups that the user should be considered a member of. Typically this involves filtering groups by looking for the user's DN on a group attribute. The *subtree search scope* determines whether the search should include the subtree extending from the *search context*.

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 and HTTP 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.

By default, this authentication provider caches the result of an authentication for a short period of time. This reduces the load on the Directory service if the same credentials are presented frequently within a short period of time. The length of time a result will be cached is defined by context variable `qpid.auth.cache.expiration_time` (default to 600 seconds). The cache can be disabled by setting the context variable `qpid.auth.cache.size` to 0.

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).

8.1.2.1. SPNEGO Authentication

SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) based authentication can be configured for Web Management Console and REST API.

A special JAAS login configuration needs to be provided for Service Principal Name (SPN) *HTTP/{FQDN}@REALM* in addition to configuration provided for broker service principal in scope *com.sun.security.jgss.accept*. An example of such SPNEGO configuration is provided below,

```
spnego {
    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-for-HTTP-principal"
    principal="HTTP/broker.org";
};
```

Important

Please, note that in the example above a principal name is specified as *HTTP/broker.org* where *broker.org* is supposed to be a fully qualified name of the host where broker is running. The

FQDN used to access the Broker must match the host name in the SPN exactly otherwise the authentication will fail.

A name of configuration module in the example above is *spnego*. It can be communicated to the Kerberos authentication provider via context variable or JVM system property *qpid.auth.gssapi.spnegoConfigScope*. For example,

```
export QPID_OPTS=-Dqpid.auth.gssapi.spnegoConfigScope=spnego -Djavax.secur
```

The RELM part in name of authenticated principal logged with SPNEGO mechanism can be stripped by setting context variable *qpid.auth.gssapi.spnegoStripRealmFromPrincipalName* to *true*.

8.1.3. OAuth2

This authentication provider allows users to login to the broker using credentials from a different service supporting OAuth2. Unfortunately, the OAuth2 specification [https://www.rfc-editor.org/rfc/rfc6749.txt] does not define a standard why to get the identity of a subject from an access token. However, most OAuth2 implementations provide such functionality, although in different ways. Qpid handles this by providing so called IdentityResolvers. Currently the following services are supported:

- CloudFoundry
- Facebook
- GitHub
- Google
- Microsoft Live

Since all of these, with the exception of CloudFoundry, are tied to a specific service they come with defaults for the Scope, Authorization-, Token-, and IdentityResolverEndpoint.

By default, this authentication provider caches the result of an authentication for a short period of time. This reduces the load on the OAuth2 service if the same token is presented frequently within a short period of time. The length of time a result will be cached is defined by context variable `qpid.auth.cache.expiration_time` (default to 600 seconds). The cache can be disabled by setting the context variable `qpid.auth.cache.size` to 0.

8.1.4. 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 and will treat any non-sasl authentication processes on these ports as successful with the given username.

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.

8.1.5. 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.

8.1.6. 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 8.5, “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.

8.1.7. 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 8.5, “Configuration Encryption” must be used.

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

8.1.8. 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.

8.1.8.1. 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
```

8.1.9. 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 8.5, “Configuration Encryption”.

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

8.1.10. 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.

8.1.10.1. 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.

8.1.11. Composite Provider

Composite Provider uses existing username / password authentication providers allowing to perform authentication against them in order defined. It can contains following authentication providers:

- *MD5 Provider*
- *Plain Provider*
- *SCRAM SHA Providers*
- *Simple LDAP Providers*

When performing authentication, composite provider checks presence of a user with a given username in the first delegate provider and if found, performs authentication. It should be considered, that in case of name collision (when delegate providers contains users with same username but different passwords), it's not guaranteed that authentication will succeed even with the correct credentials. Therefore username collision should be avoided, i.e. each delegate provider should contain users with unique usernames.

Table 8.1. SASL Mechanisms

Authentication provider	**SASL mechanisms**
MD5 Provider	PLAIN, CRAM-MD5-HASHED, CRAM-MD5-HEX
Plain	PLAIN, CRAM-MD5, SCRAM-SHA-1, SCRAM-SHA-256

Authentication provider	SASL mechanisms
SCRAM SHA Providers	PLAIN, SCRAM-SHA-1, SCRAM-SHA-256
Simple LDAP Providers	PLAIN

Composite provider exposes intersection of SASL mechanism provided by its delegates.

8.2. Group Providers

The Apache Qpid Broker-J utilises GroupProviders to allow assigning users to groups for use in ACLs or CLTs. 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.

8.2.2. ManagedGroupProvider

The *ManagedGroupProvider* allows specifying group membership as part of broker configuration. In future version of Brokers GroupFile Provider will be replaced by this one.

8.2.3. CloudFoundryDashboardManagementGroupProvider

The *CloudFoundryDashboardManagementGroupProvider* allows mapping of service instance ids to qpid management groups.

One use case is restricting management capabilities of a OAuth2 authenticated user to certain virtual hosts. For this, one would associate a cloudfoundry service id with each virtual host and have an ACL with a separate management group for each virtual host. Given the correct service instance id to

management group mapping the GroupProvider will then associate the user with each management group the user is provisioned to manage the associated service instance in the CloudFoundry dashboard [http://docs.cloudfoundry.org/services/dashboard-sso.html#checking-user-permissions].

8.3. Access Control Providers

The Access Control Provider governs the actions that a user may perform.

There are two points within the hierarchy that enforce access control: the Broker itself and at each Virtual Host. When an access decision needs to be made, the nearest control point configured with a provider is consulted for a decision. The example, when making a decision about the ability to say, consume from, a Queue, if the Virtual Host is configured with Access Control Provider it is consulted. Unless a decision is made, the decision is delegated to the Access Control Provider configured at the Broker.

Access Control Providers are configured with a list of ACL rules. The rules determine to which objects the user has access and what actions the user may perform on those objects. Rules are ordered and are considered top to bottom. The first matching rule makes the access decision.

ACL rules may be written in terms of user or group names. A rule written in terms of a group name applies to the user if he is a member of that group. Groups information is obtained from the Authentication Providers and Group Providers. Writing ACL in terms of groups is recommended.

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

8.3.1. Types

There are currently two types of Access Control Provider implementing ACL rules.

- *RulesBased* - a provider that stores the rules-set within the Broker's or VirtualHost's configuration. When used with HA, the Virtualhost rules automatically propagated to all nodes participating within the HA group.
- *ACLFile* - an older provider that references an externally provided ACL file (or data url). This provider is deprecated.

8.3.2. ACL Rules

An ACL rule-set is an ordered list of ACL rules.

An ACL rule comprises matching criteria that determines if a rule applies to a situation and a decision outcome. The rule produces an outcome only if the all matching criteria are satisfied.

Matching criteria is composed of an ACL object type (e.g. `QUEUE`), an ACL action (e.g. `UPDATE`) and other properties that further refine if a match is made. These properties restrict the match based on additional criteria such as name or IP address. ACL Object type `ALL` matches any object. Likewise ACL Action `ALL` matches any action.

Let's look at some examples.

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

As discussed, the ACL rule-set is considered in 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 action 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
```

If a rule-set fails to make a decision, the result is configurable. By default, the `RuleBased` provider defers the decision allowing another provider further up the hierarchy to make a decision (i.e. allowing the VirtualHost control point to delegate to the Broker). In the case of the ACLFile provider, by default, its rule-set implicit have a rule denying all operations to all users. It is as if the rule-set ends with `ACL DENY ALL ALL`. If no access control provider makes a decision the default is to deny the action.

When writing a new ACL, a useful approach is to begin with an rule-set containing only

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

at the Broker control point which will cause the Broker to deny all operations with details of the denial logged. 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.

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 abstractions and apply them consistently across the whole system.

8.3.3. Syntax

ACL rules follow this syntax:

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

The <property-values> can be a single value property="single value" or a list of comma separated values in brackets property=["value1", "value2", "value3"]. If a property repeats then it will be interpreted as list of values, for example name="n1" name="n2" name="n3" is interpreted as name=["n1", "n2", "n3"].

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 8.2. 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 8.3. 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, 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
ACCESS	Applied when a connection is made for messaging or management	VIRTUALHOST, MANAGEMENT	name (for VIRTUALHOST only)
BIND	Applied when queues are bound to exchanges	EXCHANGE	name, routingKey, queue_name, virtualhost_name, temporary, durable
UNBIND	Applied when queues are unbound from exchanges	EXCHANGE	name, routingKey, queue_name, 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 the contents of a queue is purged	QUEUE	
UPDATE	Applied when an object is updated	VIRTUALHOSTNODE, VIRTUALHOST, EXCHANGE, QUEUE, USER, GROUP	see EXCHANGE and QUEUE properties
CONFIGURE	Applied when a Broker/ Port/Authentication	BROKER	

Action	Description	Supported object types	Supported properties
	Provider/Access Control Provider/BrokerLogger is created/update/ deleted.		
ACCESS_LOGS	Allows/denies the specific user to download log file(s).	BROKER, VIRTUALHOST	name (for VIRTUALHOST only)
SHUTDOWN	Allows/denies the specific user to shutdown the Broker.	BROKER	
INVOKE	Allows/denies the specific user to invoke the named operation.	BROKER, VIRTUALHOSTNODE, VIRTUALHOST, EXCHANGE, QUEUE, USER, GROUP	method_name, name and virtualhost_name

Table 8.4. List of ACL objects

Object type	Description	Supported actions	Supported properties	Allowed in Virtualhost ACLs?
VIRTUALHOSTNODE	A virtualhostnode or remote replication node	ALL, CREATE, UPDATE, DELETE, INVOKE	name	No
VIRTUALHOST	A virtualhost	ALL, CREATE, UPDATE, DELETE, ACCESS, ACCESS_LOGS, INVOKE	name, connection_limit, connection_frequency_limit	No
QUEUE	A queue	ALL, CREATE, DELETE, PURGE, CONSUME, UPDATE, INVOKE	name, autodelete, temporary, durable, exclusive, alternate, owner, virtualhost_name	Yes
EXCHANGE	An exchange	ALL, ACCESS, CREATE, DELETE, BIND, UNBIND, PUBLISH, UPDATE, INVOKE	name, autodelete, temporary, durable, type, virtualhost_name, queue_name(only for BIND and UNBIND), routingkey(only for BIND and UNBIND, PUBLISH)	Yes
USER	A user	ALL, CREATE, DELETE,	name	No

Object type	Description	Supported actions	Supported properties	Allowed in Virtualhost ACLs?
		UPDATE, INVOKE		
GROUP	A group	ALL, CREATE, DELETE, UPDATE, INVOKE	name	No
BROKER	The broker	ALL, CONFIGURE, ACCESS_LOGS, INVOKE		No

Table 8.5. List of ACL properties

name	String. Object name, such as a queue name or exchange 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, or fanout
alternate	String. Name of the alternate exchange
queue_name	String. Name of the queue (used only when the object is EXCHANGE for BIND and UNBIND actions)
component	String. 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/8/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.
method_name	String. The name of the method. A trailing wildcard (*) is permitted. Used with INVOKE ACL action.
attribute_names	Specifies attribute name criteria. Used by UPDATE ACL actions only. Rules with this criteria will match if and only if the set of attributes being updated Comma separated list of attribute names . This criteria will match if all attributes included within the update appear in the set described by `attribute_names`.

8.3.4. Worked Examples

Here are some example ACLs illustrating common use cases.

8.3.4.1. 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 actions. Neither `operator` nor `readonly` should be allowed to connect clients for messaging.

Example 8.1. Worked example 1 - Management rights

```
# Deny operator/readonly permission to connect for messaging.
ACL DENY-LOG operator ACCESS VIRTUALHOST
ACL DENY-LOG readonly ACCESS VIRTUALHOST
# Give operator permission to perfom all actions
ACL ALLOW operator ALL ALL
# Give readonly access permission to virtualhost. (Read permission for a
```

```
ACL ALLOW readonly ACCESS MANAGEMENT
...
... rules for other users
...
```

8.3.4.2. Worked example 2 - Simple Messaging

Suppose you wish to permission a system for application messaging. User `publisher` needs permission to publish to `appqueue` and consumer needs permission to consume from the same queue object. We also want `operator` to be able to inspect messages and delete messages in case of the need to intervene. This example assumes that the queue exists on the Broker.

We use this ACL to illustrate separate Broker and Virtualhost access control providers.

The following ACL rules are given to the Broker.

Example 8.2. Worked example 2a - Simple Messaging - Broker ACL rules

```
# This gives the operate permission to delete messages on all queues on all virtua
ACL ALLOW operator ACCESS MANAGEMENT
ACL ALLOW operator INVOKE QUEUE method_name="deleteMessages"
ACL ALLOW operator INVOKE QUEUE method_name="getMessage*"
```

Example 8.3. Worked example 2b - Simple Messaging - Broker ACL rules with multi-value property

```
# This gives the operate permission to delete messages on all queues on all virtua
ACL ALLOW operator ACCESS MANAGEMENT
ACL ALLOW operator INVOKE QUEUE method_name=["deleteMessages", "getMessage*"]
```

And the following ACL rule-set is applied to the Virtualhost. The default outcome of the Access Control Provider must be `DEFERED`. This means that if a request for access is made for which there are no matching rules, the decision will be deferred to the Broker so it can make a decision instead.

Example 8.4. Worked example 2 - Simple Messaging - Virtualhost ACL rules

```
# Configure the rule-set to DEFER decisions that have no matching rules.
CONFIG DEFAULTDEFER=TRUE
# Allow client and server to connect to the virtual host.
ACL ALLOW publisher ACCESS VIRTUALHOST
ACL ALLOW consumer ACCESS VIRTUALHOST

ACL ALLOW publisher PUBLISH EXCHANGE name="" routingKey="appqueue"
ACL ALLOW consumer CONSUME QUEUE name="appqueue"
# In some addressing configurations, the Qpid JMS AMQP 0-x client, will declare th
# The following line allows for this.  For the Qpid JMS AMQP 1.0 client, this is n
ACL ALLOW consumer CREATE QUEUE name="appqueue"
```

8.3.4.3. Worked example 3 - 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.

Example 8.5. Worked example 3 - firewall-like access control

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

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

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

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

# Deny access to all users in the IP ranges 192.168.1.0-192.168.1.255 an
# using the notation specified in RFC 4632, "Classless Inter-domain Rout
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 an
# using wildcard notation.
ACL DENY-LOG messaging-users ACCESS VIRTUALHOST \
from_network="192.169.1.*,192.169.2.*"
```

8.4. Connection Limit Providers

The Connection Limit Provider governs the limits of connections that an user can simultaneously open.

There are two points within the hierarchy that enforce connection limits: the Broker itself and at each Virtual Host. When a limit needs to be checked, every check point configured with a provider is consulted for a decision. The example, when making a decision about the opening a new connection. If the Virtual Host is configured with Connection Limit Provider then the limits are checked. Unless the connection is rejected, the decision is delegated to the Connection Limit Provider configured at the Broker.

Connection Limit Provider is configured with a set of CLT (connection limit) rules. The rules determine the limit of open connections, how many connections can user open on the AMQP Ports.

CLT rules may be written in terms of user or group names. A rule written in terms of a group name applies to the user if he is a member of that group. Groups information is obtained from the Authentication Providers and Group Providers. Writing CLT rules in terms of user names is recommended.

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

8.4.1. Types

There are currently two types of Connection Limit Provider implementing CLT rules.

- *RulesBased* - a provider that stores the rule-set within the Broker's or VirtualHost's configuration.
- *ConnectionLimitFile* - a provider that references an externally provided CLT file (or data url).

8.4.2. Connection Limit Rules

An CLT rule is composed of an user or group identification, AMQP port name and connection limits. Let's look at some example.

```
# Limits simultaneously open connection by alice on brokerAmqp port up
CLT alice port=brokerAmqp connection_count=10
```

If there is multiple rules for given user (or group) then the rules are merge into a single most restrictive rule.

```
CLT alice port=brokerAmqp connection_count=10
CLT alice port=brokerAmqp connection_count=12 connection_frequency_cou
CLT alice port=brokerAmqp connection_frequency_count=100/1m
```

The previous rules will be merge into a single effective rule.

```
CLT alice port=brokerAmqp connection_count=10 connection_frequency_cou
```

The rules are applied in following order:

1. The effective rule for given user.
2. The effective rule for given set of groups that user is a member of.
3. The default rule, a rule with the user ALL that matches any user.

At the first broker looks for a rule for given user. If any rule is not found then broker will look for the group rules. If any group rule is not found then broker will look for a default rule. An user without any rule is not restricted.

8.4.3. Syntax

Connection limit rules follow this syntax:

```
CLT {<user-name>|<group-name>|ALL} [BLOCK] [port=<AMQP-port-name>|ALL]
```

A rule with user name ALL is default rule. Likewise a rule with port=ALL is applied to all ports. The parameter BLOCK is optional and marks user or group that is not allowed to connect on the port.

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

```
# A comment
CLT alice port=brokerAMQP connection_limit=10 # Also a comment
CLT mark port=brokerAMQP \ # A broken line
connection_limit=10 \
connection_frequency_limit=60/1m
CLT ALL BLOCK # A default rule
```

Table 8.6. List of connection limit (CLT) properties

connection_limit	Integer. A maximum number of connections the messaging user can establish to the Virtual Host on AMQP port. Alternatives: connection-limit, connectionLimit.
connection_frequency_limit	A maximum number of connections the messaging user can establish to the Virtual Host on AMQP port within defined period of time, which is 1 minute by default. The connection frequency limit is specified in the format: limit/period, where time period is written as xHyMz.wS - x hours, y minutes and z.w seconds. In case of time period 1 hour/minute/second the digit 1 can be omitted, for example: 7200/H or 120/M or 2/S. (7200/H is not the same frequency limit as 120/H or 2/S). If the period is omitted then the default frequency period is used. If required, the default frequency period can be changed using CONFIG command. See an example below. Setting it to zero or negative value turns off the connection frequency evaluation. Alternatives: connection-frequency-limit, connectionFrequencyLimit.
port	String. The AMQP port name, ALL is the default value.

The default time period for frequency limit can be set up with the `CONFIG` command. Default frequency period is specified in ms.

```
CONFIG default_frequency_period=60000
```

default-frequency-period and defaultFrequencyPeriod are valid alternatives to the default_frequency_period.

The default frequency period may be specified as context variable `qpid.broker.connectionLimiter.frequencyPeriodInMillis`.

The Broker logs rejected connections when an user breaks the limit. But the Broker could also log the accepted connections with current counter value. The full logging could be turn on with `CONFIG` command.

```
CONFIG log_all=true default_frequency_period=60000
```

log-all and logAll are valid alternatives to the log_all.

8.4.4. Worked Example

Here are some example of connection limits illustrating common use cases.

Suppose you wish to restrict two users: a user `operator` can establish at the most 50 connections on any port. A user `publisher` can establish 30 new connection per two minutes but at the most 20 parallel connections on `amqp` port. Another users should be blocked.

Example 8.6. CLT file example

```
# Limit operator
CLT operator connection_limit=50
# Limit publisher
CLT publisher port=amqp connection_frequency_limit=30/2M connection_limi
# Block all users by default
CLT ALL BLOCK
```

8.5. 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 configuration in a form that is difficult to read.

The Broker ships with an encryptor implementations called `AESGCMKeyFile` and `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.

Important

AESKeyFile encryptor is considered as not safe, it is deprecated and will be removed in one of the next releases. AESGCMKeyFile encryptor should be used instead.

Important

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

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

8.5.1. Configuration

The `AESGCMKeyFile` or `AESKeyFile` encryptor providers are 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.

8.5.2. Alternate Implementations

If the `AESGCMKeyFile` 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. Logging

This section describes the flexible logging capabilities of the Apache Qpid Broker-J.

- The Broker is capable of sending logging events to a variety of destinations including plain files, remote syslog daemons, and an in-memory buffer (viewable from Management). The system is also open for extension meaning it is possible to produce a plugin to log to a bespoke destination.
- Logging can be dynamically configured at runtime. For instance, it is possible to temporarily increase the logging verbosity of the system whilst a problem is investigated and then revert later, all without the need to restart the Broker.
- Virtualhosts can be configured to generate their own separate log, and the Broker is capable of generating a log either inclusive or exclusive of virtualhost events.
- Logs are accessible over Management, removing the need for those operating the Broker to have shell level access.

In the remainder of this section you will first find a description of the concepts used in the logging subsystem. Next, you find a description of the default configuration. The section then concludes with a in-depth description of the loggers themselves and how they may be configured.

9.1.1. Concepts

The logging subsystem uses two concepts:

- A *Logger* is responsible for production of a log. The Broker ships a variety of loggers, for instance, a file logger, which is capable of writing a log file to the file system, a Syslog Logger capable of writing to a remote syslog daemon and console logger capable of writing to stdout or stderr.

  Loggers are attached at two points within the Broker Model; the Broker itself and the virtualhosts. Loggers attached at the Broker can capture log events for the system as a whole, or can exclude events related to virtualhosts.

  Loggers attached to a virtualhost capture log events relating to that virtualhost only.

  The Broker and virtualhosts can have zero or more Loggers. If no loggers are configured, no logging is generated at all.

- *Inclusion rules* govern what appears within a log. Inclusion rules are associated with Loggers. This means it is possible for different Loggers to have different contents.

  A Logger with no inclusion rules will produce an empty log.

9.1.2. Default Configuration

The default configuration is designed to be suitable for use without change in small production environments. It has the following characteristics:

- The Broker generates a single log file `qpid.log`. This logfile is rolled automatically when the file reaches 100MB. A maximum history of one file is retained. On restart the the log will be appended to.

  The log contains:

- All operational logging events. See Appendix C, *Operational Logging*.
- Log events from Qpid itself deemed informational or higher.
- Log events from Qpid's dependencies (such as Derby or Jetty) that are deemed warning or higher.

The default location for the log file is `${QPID_WORK}/log/qpid.log`.

- The Broker also caches the last 4096 log events in a memory cache. By default, the memory logger logs the same things the file logger does.

The configuration can be customised at runtime using Management. This makes it possible to investigate unusual conditions *without* the need to restart the Broker. For instance, you may alter the logging level so that a verbose log is produced whilst an investigation is in progress and revert the setting later, all without the need to restart the Broker.

9.1.3. Loggers

Loggers are responsible for the writing of a log. The log includes log events that match a Logger's inclusion rules.

Loggers are associated with either the Broker or a virtualhost. Virtualhost loggers write only log events related to that virtualhost. Broker Loggers write log events from the Broker as a whole. Optionally a Broker Logger can be configured to exclude log events coming from virtualhosts. These abilities can be usefully exploited together in managed service scenarios to produce separate logs for separate user groups.

Loggers can be added or removed at runtime, without restarting the Broker. However changes to a Logger's configuration such as filenames and rolling options don't take effect until the next restart. Changes to a Logger's inclusion rules take effect immediately.

All loggers allow the log event layout to be customised. Loggers understand Logback Classic Pattern Layouts [http://logback.qos.ch/manual/layouts.html#ClassicPatternLayout].

The following sections describes each Logger implementation in detail.

9.1.3.1. FileLogger

A *FileLogger* - writes a log file to the filesystem. The name and location of the log file, the rolling configuration, and compression options can be configured.

The *roll daily* option, if enabled, will cause the log file will be rolled at midnight local time. The rolled over file will have a suffix in the form `yyyy-mm-dd`. In roll daily mode, *maximum number of rolled files* controls the maximum number of *days* to be retained. Older files will be deleted.

The *maximum file size* option limits the size of any one log file. Once a log file reaches the given size, it will be rolled. The rolled over file will have the numeric suffix, beginning at `1`. If the log file rolls again, first the existing file with the suffix `.1` is renamed to `.2` and so forth. If roll daily is not in use, *maximum number of rolled files* governs the number of rolled *files* that will be retained.

Roll on restart governs whether the log file is rolled when the Broker is restarted. If not ticked, the Broker will append to the existing log file until it needs to be rolled.

9.1.3.2. ConsoleLogger

ConsoleLogger - writes a log file standard out or standard error.

9.1.3.3. SyslogLogger

SyslogLogger - writes a log file to a syslog daemon using the USER facility. The hostname and port number of the syslog daemon can be configured.

Log entries can be prefixed with a string. This string defaults to include the word `Qpid` and the name of the Broker or virtualhost. This serves to distinguish the logging generated by this Qpid instance, from other Qpid instances, or other applications using the USER.

9.1.3.4. MemoryLogger

MemoryLogger - writes a log file to a circular in-memory buffer. By default the circular buffer holds the last 4096 log events. The contents of the buffer can be viewed via Management. See Figure 9.3, “Viewing a memory logger”

9.1.3.5. GraylogLogger

GraylogLogger - sends log messages to a Graylog server in GELF format [https://docs.graylog.org/en/3.2/pages/gelf.html] via TCP. The hostname and port number of the Graylog server has to be configured. The content of the log messages is also configurable.

The logger is implemented on top of LGPL licenced library `de.siegmar:logback-gelf`. The LGPL license is incompatible with Apache License. Thus, the Graylog integration module is not included into standard broker distribution. It has to be built using option `-Dgraylog`. The built jar `org.apache.qpid:qpid-broker-plugins-graylog-logging-logback` and `de.siegmar:logback-gelf` jar should be copied manually under broker `lib` folder. The broker restart is required to have *GraylogLogger* available for the use.

9.1.4. Inclusion Rules

A *Logger* has one or more *inclusion rules*. These govern what appears in the log. A Logger with no inclusion rules will log nothing.

Inclusion rules can be added, removed or changed at runtime. Changes take place immediately.

- The *Name And Level* inclusion rule accepts log events that match a given *log event source name* and have a level that equals or exceeds the specified value.

  The log event source name refers to the fully qualified class name from which the event originates. These names permit a trailing wild card `.*`. For instance a source name of `org.apache.qpid.*` will match all events from classes in the package `org.apache.qpid` and any sub packages beneath.

  The *Level* governs the level of the events that will be included in the log. It may take one of the following values: ERROR, WARN, INFO, DEBUG, TRACE where ERROR is considered the highest and TRACE the lowest. In addition, there are two special values: OFF and ALL, the former excludes all log events whereas the latter will include everything. When considering whether a logging event should be included in the log, the logging event must have a level that matches that of the inclusion rule or be higher, otherwise the log event will not appear in the log.

9.1.5. Logging Management

The logging subsystem can be completely managed from the Web Management Console or the REST API. You can:

- Add, remove, or change the configuration of Loggers.
- Add, remove, or change the Inclusion Rules.
- For FileLoggers, download the log file and rolled log files associated with the Logger.
- For MemoryLoggers, view the last n log events

The figure that follows shows a FileLogger. The attributes area shows the configuration of the Logger. The inclusion rule table shows the rules that are associated with the Logger. The area towards the bottom of the tab allows the log files to be downloaded to the browser.

Figure 9.1. Viewing a file logger

The figure below shows the editing of the level of an inclusion rule.

Figure 9.2. Editing an inclusion rule

The figure below shows a Memory Logger. Note that the Memory Logger provides access to the cached message via the viewer towards the bottom on the tab.

Figure 9.3. Viewing a memory logger

9.2. Disk Space Management

9.2.1. Disk quota-based flow control

The Apache Qpid Broker-J supports a flow control mechanism which is 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.

The Broker will also impose flow control if the filesystem hosting a virtualhost exceeds a configured percentage..

Note

The *Producer Flow Control* can be configured on individual queue using *Producer Flow Control* overflow policy. For more details, please readSection 4.7.5, “Controlling Queue Size”.

9.2.1.1. 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 co
```

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 conditio
```

9.3. Transaction Timeout

9.3.1. General Information

The transaction timeout mechanism is used to control broker resources when clients using transactions hang, become unresponsive, or simply (due to programming error) begin a transaction and keep using it without ever calling committing or rolling back.

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. 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.

9.3.3.1. 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 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.

```
CON-1011 : Idle Transaction : 13,116 ms
CON-1011 : Idle Transaction : 14,116 ms
CON-1011 : Idle Transaction : 15,118 ms
CON-1002 : Close : Idle transaction timed out
```

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

```
CON-1010 : Open Transaction : 12,406 ms
CON-1010 : Open Transaction : 13,406 ms
CON-1010 : Open Transaction : 14,406 ms
CON-1002 : Close : Open transaction timed out
```

9.3.3.2. Client Side Effect

After a Close threshold has been exceeded, the Broker will close the client's connection. The application must reconnect itself in order to continue work. If the client is a JMS client, the application will be notified by the exception listener. [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html]

9.3.4. Configuration

9.3.4.1. 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 closure timeouts, it should be noted that a timeout on any producer or consumer will cause the connection to be closed - this disconnecting all producers and consumers created on that connection.

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 an attribute 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 via the queue's *alternate binding* (if one has been defined), or will discard the message.

When using AMQP 1.0 the current delivery count of a message is available to the consuming application via the `message-count` message header (exposed via the `JMSXDeliveryCount` JMS message property when using JMS). When using the AMQP 0-8..0-10 protocols this information is not available.

Note

When using AMQP 0-8..0-10, in order for a maximum delivery count to be enforced, the consuming application *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 application 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 Qpid JMS Client 0-x and using AMQP 0-8, 0-9, or 0-9-1 protocols, it is necessary to set the client system property `qpid.reject.behaviour` or connection or binding URL option `rejectbehaviour` to the value `server`.

9.4.3. Alternate Binding

Once the maximum delivery count is exceeded, if the queue has an `alternateBinding` specified, the Broker automatically routes the message via the alternate binding. The alternate binding would normally specify a queue designated for that purpose of receiving the undeliverable messages. By convention such queues are known as dead-letter queues or simply DLQs.

It is possible to configure the broker to automatically default a DLQ for every queue created. To do this one can set the context variable `queue.defaultAlternateBinding` at the Virtual Host (or above) level. For example, by setting the value to `{\"destination\": \"$${this:name}_DLQ\"}` a new queue *exampleQueue* will default to having an alternate binding to *exampleQueue_DLQ*. To avoid error this should be combined with setting a node auto creation policy on the VirtualHost, so that such DLQs are automatically created, e.g.

```
"nodeAutoCreationPolicies" : [ {
"pattern" : ".*_DLQ",
"nodeType" : "Queue",
"attributes" : {
"alternateBinding" : ""
},
"createdOnPublish" : true,
"createdOnConsume" : true
} ]
```

Note

For the autocreated DLQs it is important to override the default alternate binding, as above, else the creation of an infinite chain of DLQs for DLQs will be attempted.

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, the Apache Qpid JMS AMQP 0-x clients 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 Apache Qpid Broker-J 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.

By default, the Apache Qpid JMS AMQP 0-x produces mandatory messages when using queue destinations. Topic destinations produce 'non-mandatory' messages.

9.5.2. Configuring *closeWhenNoRoute*

The Port 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.

See the Qpid JMS AMQP 0-x client documentation [../../jms-client-0-8/book/JMS-Client-0-8-Connection-URL.html] for details of enabling this feature client side.

9.6. Flow to Disk

Flow to disk limits the amount of direct and 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 75% of the max direct 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 direct memory with the default of 75% available for messages. The first queue will have a target size of 320MB and the second 430MB. Once 750MB 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 750MB.

Flow to disk is configured by Broker context variable `broker.flowToDiskThreshold`. It is expressed as a size in bytes and defaults to 75% of the JVM maximum heap size.

9.6.1. Flow to Disk Monitoring

A number of statistics attributes are available on the `Broker` to allow monitoring of the amount of utilized direct memory by the enqueued messages.

The total amount of allocated direct memory by the `Broker` can be determined by checking `Broker` statistics `usedDirectMemorySize`. There is another `Broker` level statistics `directMemoryTotalCapacity` to get the total amount of allocated direct memory. Usually, the values reported by both statistics attributes `usedDirectMemorySize` and `directMemoryTotalCapacity` are the same or do not differ much.

The direct memory consumed by the `VirtualHost` messages is reported as `VirtualHost` statistics `inMemoryMessageSize`. The current value of `VirtualHost` direct memory threshold is exposed with statistics attribute `inMemoryMessageThreshold`. When the value of `inMemoryMessageSize` is greater than `inMemoryMessageThreshold`, the flow to disk is triggered to bring the amount of direct memory consumed by the `VirtualHost` messages in-line with the `inMemoryMessageThreshold`.

9.6.2. Flow to Disk Logging

The `Flow to Disk` events are not reported as operational logs or `INFO` logs due to quite frequent triggering of `Flow to Disk` for messaging use cases requiring holding messages on the `Broker` side for some time. As result, the `Flow to Disk` logs can quickly dominate the broker logs and cause unnecessary disk consumption.

Though, if required, the `Flow to Disk` DEBUG logs can be enabled by adding the following logging rule into the corresponding `Broker` logger.

Example 9.1. Flow to Disk logging rule

```
{
  "name" : "DirectMemory",
  "type" : "NameAndLevel",
  "level" : "DEBUG",
  "loggerName" : "org.apache.qpid.server.directMemory.*"
}
```

Please note, that the logger `org.apache.qpid.server.directMemory.broker` is used by the `Broker` to report conditions when direct memory utilization exceeds the pred-defined `Broker` threshold, whilst the logger `org.apache.qpid.server.directMemory.virtualhost` is used to report conditions when direct memory utilization by the `VirtualHost` messages exceeds the current value of the `VirtualHost` threshold.

9.7. Consumers

A Consumer is created when an AMQP connection wishes to receive messages from a message source (such as a Queue). The standard behaviours of consumers are defined by the respective AMQP specification, however in addition to the standard behaviours a number of Qpid specific enhancements are available

9.7.1. Priority

By default, when there are multiple competing consumers attached to the same message source, the Broker attempts to distribute messages from the queue in a "fair" manner. Some use cases require allocation of messages to consumers to be based on the "priority" of the consumer. Where there are multiple consumers

having differing priorities, the Broker will always attempt to deliver a message to a higher priority consumer before attempting delivery to a lower priority consumer. That is, a lower priority consumer will only receive a message if no higher priority consumers currently have credit available to consume the message, or those consumers have declined to accept the message (for instance because it does not meet the criteria of any selectors associated with the consumer).

Where a consumer is created with no explicit priority provided, the consumer is given the highest possible priority.

9.7.1.1. Creating a Consumer with a non-standard priority

In AMQP 0-9 and 0-9-1 the priority of the consumer can be set by adding an entry into the table provided as the `arguments` field (known as the `filter` field on AMQP 0-9) of the `basic.consume` method. The key for the entry must be the literal short string `x-priority`, and the value of the entry must be an integral number in the range -2^{31} to $2^{31}-1$.

In AMQP 0-10 the priority of the consumer can be set in the map provided as the `arguments` field of the `message.subscribe` method. The key for the entry must be the literal string `x-priority`, and the value of the entry must be an integral number in the range -2^{31} to $2^{31}-1$.

In AMQP 1.0 the priority of the consumer is set in the `properties` map of the `attach` frame where the broker side of the link represents the sending side of the link. The key for the entry must be the literal string `priority`, and the value of the entry must be an integral number in the range -2^{31} to $2^{31}-1$.

When using the Qpid JMS client for AMQP 0-9/0-9-1/0-10 the consumer priority can be set in the address being used for the Destination object.

Table 9.1. Setting the consumer priority

Syntax	Example
Addressing	myqueue : { link : { x-subscribe: { arguments : { x-priority : '10' } } } }
Binding URL	direct://amq.direct/myqueue/myqueue?x-qpid-replay-priority='10'

9.8. 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.

9.9. Message Compression

The Apache Qpid Broker-J 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.

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.

9.10. 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.

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

9.11. Memory

9.11.1. Introduction

Understanding how the Qpid broker uses memory is essential to running a high performing and reliable service. A wrongly configured broker can exhibit poor performance or even crash with an `OutOfMemoryError`. Unfortunately, memory usage is not a simple topic and thus requires some in depth explanations. This page should give the required background information to make informed decisions on how to configure your broker.

Section 9.11.2, “Types of Memory” explains the two different kinds of Java memory most relevant to the broker. Section 9.11.3, “Memory Usage in the Broker” goes on to explain which parts of the broker use what kind of memory. Section 9.11.4, “Low Memory Conditions” explains what happens when the system runs low on memory. Section 9.11.5, “Defaults” lays out the default settings of the Qpid broker. Finally, Section 9.11.6, “Memory Tuning the Broker” gives some advice on tuning your broker.

9.11.2. Types of Memory

While Java has a couple of different internal memory types we will focus on the two types that are relevant to the Qpid broker. Both of these memory types are taken from the same physical memory (RAM).

9.11.2.1. Heap

Normally, all objects are allocated from Java's heap memory. Once, nothing references an object it is cleaned up by the Java Garbage Collector and it's memory returned to the heap. This works fine for most use cases. However, when interacting with other parts of the operating system using Java's heap is not ideal. This is where the so called direct memory comes into play.

9.11.2.2. Direct

The world outside of the JVM, in particular the operating system (OS), does not know about Java heap memory and uses other structures like C arrays. In order to interact with these systems Java needs to copy data between its own heap memory and these native structures. This can become a bottle neck when there is a lot of exchange between Java and the OS like in I/O (both disk and network) heavy applications. Java's solution to this is to allow programmers to request `ByteBuffer`s from so called direct memory. This is an opaque structure that *might* have an underlying implementation that makes it efficient to interact with the OS. Unfortunately, the GC is not good at tracking direct memory and in general it is inadvisable to use direct memory for regular objects.

9.11.3. Memory Usage in the Broker

This section lists some note worthy users of memory within the broker and where possible lists their usage of heap and direct memory. Note that to ensure smooth performance some heap memory should remain unused by the application and be reserved for the JVM to do house keeping and garbage collection. Some guides [https://docs.oracle.com/cd/E17277_02/html/java/com/sleepycat/je/util/DbCacheSize.html] advise to reserve up to 30% of heap memory for the JVM.

9.11.3.1. Broker

The broker itself uses a moderate amount of heap memory ($\approx$15 MB). However, each connection and session comes with a heap overhead of about 17 kB and 15 kB respectively. In addition, each connection reserves 512 kB direct memory for network I/O.

9.11.3.2. Virtual Hosts

The amount of memory a Virtual Host uses depends on its type. For a JSON Virtual Host Node with a BDB Virtual Host the heap memory usage is approximately 2 MB. However, each BDB Virtual Hosts has a mandatory cache in heap memory which has an impact on performance. See below for more information.

9.11.3.3. Messages

Messages and their headers are kept in direct memory and have an additional overhead of approximately 1 kB heap memory each. This means that most brokers will want to have more direct memory than heap memory. When many small messages accumulate on the broker the 1 kB heap memory overhead can become a limiting factor.

When the broker is running low on direct memory it will evict enqueued messages from memory and flow them to disk. For persistent messages this only means freeing the direct memory representation because they always have an on-disk representation to guard against unexpected failure (e.g., a power cut). For transient messages this implies additional disk I/O. After being flowed to disk messages need to be re-read from disk before delivery.

Please, note that messages from uncommitted transactions are not flowed to disk as part of running into low direct memory conditions, as they are not enqueued yet. The `Connection` has its own threshold for keeping messages from uncommitted transactions in memory. Only when `Connection` threshold is breached, the uncommitted messages on the connection are flowed to disk.

9.11.3.4. Message Store

Berkeley DB (BDB)

The broker can use Oracle's BDB JE (BDB) as a message store to persist messages by writing them to a database. BDB uses a mandatory cache for navigating and organising its database structure. Sizing and tuning this cache is a topic of its own and would go beyond the scope of this guide. Suffice to say that by default Qpid uses 5% of heap memory for BDB caches (each Virtual Host uses a separate cache) or 10 MB per BDB store, whichever is greater. See the official webpage [http://www.oracle.com/us/products/database/berkeley-db/je] especially this page [http://docs.oracle.com/cd/E17277_02/html/java/com/sleepycat/je/util/DbCacheSize.html] for more information. For those interested, Qpid uses EVICT_LN [http://docs.oracle.com/cd/E17277_02/html/java/com/sleepycat/je/CacheMode.html#EVICT_LN] as its default JE cacheMode.

Derby

TODO

9.11.3.5. HTTP Management

Qpid uses Jetty for the HTTP Management (both REST and Web Management Console). When the management plugin is loaded it will allocate the memory it needs and should not require more memory during operation and can thus be largely ignored.

9.11.4. Low Memory Conditions

9.11.4.1. Low on Heap Memory

When the broker runs low on heap memory performance will degrade because the JVM will trigger full garbage collection (GC) events in a struggle to free memory. These full GC events are also called stop-

the-world events as they completely halt the execution of the Java application. Stop-the-world-events may take any where from a couple of milliseconds up to several minutes. Should the heap memory demands rise even further the JVM will eventually throw an OutOfMemoryError which will cause the broker to shut down.

9.11.4.2. Low on Direct Memory

When the broker detects that it uses 75% of available direct memory it will start flowing incoming transient messages to disk and reading them back before delivery. This will prevent the broker from running out of direct memory but may degrade performance by requiring disk I/O.

9.11.5. Defaults

By default Qpid uses these settings:

- 0.5 GB heap memory
- 1.5 GB direct memory
- 5% of heap reserved for the BDB JE cache.
- Start flow-to-disk at 75% direct memory utilisation.

As an example, this would accommodate a broker with 50 connections, each serving 5 sessions, and each session having 1000 messages of 1 kB on queues in the broker. This means a total of 250 concurrent sessions and a total of 250000 messages without flowing messages to disk.

9.11.6. Memory Tuning the Broker

9.11.6.1. Java Tuning

Most of these options are implementation specific. It is assumed you are using Oracle Java 11.

- Heap and direct memory can be configured through the `QPID_JAVA_MEM` environment variable.

9.11.6.2. Qpid Tuning

- The system property `qpid.broker.bdbTotalCacheSize` sets the total amount of heap memory (in bytes) allocated to BDB caches.
- The system property `broker.flowToDiskThreshold` sets the threshold (in bytes) for flowing transient messages to disk. Should the broker use more than direct memory it will flow incoming messages to disk. Should utilisation fall beneath the threshold it will stop flowing messages to disk.
- The system property `connection.maxUncommittedInMemorySize` sets the threshold (in bytes) for total messages sizes (in bytes) from connection uncommitted transactions when messages are hold in memory. If threshold is exceeded, all messages from connection in-flight transactions are flowed to disk including those arriving after breaching the threshold.

9.11.6.3. Formulae

We developed a simple formula which estimates the *minimum* memory usage of the broker under certain usage. These are rough estimate so we strongly recommend testing your configuration extensively. Also, if your machine has more memory available by all means use more memory as it can only improve the

performance and stability of your broker. However, remember that both heap and direct memory are served from your computer's physical memory so their sum should never exceed the physically available RAM (minus what other processes use).

$memory_{heap}$ = 15 MB + 20 kB * $N_{sessions}$ + (1.7 kB + (120 + $averageSize_{headerNameAndValue}$) * $averageNumber_{headers}$)* $N_{messages}$ + 100 kB * $N_{connections}$

$memory_{direct}$ = 2 MB + (200 B + $averageSize_{msg}$ *2)* $N_{messages}$ + 1MB * $N_{connections}$

Where N denotes the total number of connections/sessions/messages on the broker. Furthermore, for direct memory only the messages that have not been flowed to disk are relevant.

Note

The formulae assume the worst case in terms of memory usage: persistent messages and TLS connections. Transient messages consume less heap memory than persistent and plain connections consume less direct memory than TLS connections.

9.11.6.4. Things to Consider

Performance

Choosing a smaller direct memory size will lower the threshold for flowing transient messages to disk when messages accumulate on a queue. This can have impact on performance in the transient case where otherwise no disk I/O would be involved.

Having too little heap memory will result in poor performance due to frequent garbage collection events. See Section 9.11.4, “Low Memory Conditions” for more details.

OutOfMemoryError

Choosing too low heap memory can cause an OutOfMemoryError which will force the broker to shut down. In this sense the available heap memory puts a hard limit on the number of messages you can have in the broker at the same time.

If the Java runs out of direct memory it also throws a OutOfMemoryError resulting the a broker shutdown. Under normal circumstances this should not happen but needs to be considered when deviating from the default configuration, especially when changing the flowToDiskThreshold.

If you are sending very large messages you should accommodate for this by making sure you have enough direct memory.

9.12. Broker Instrumentation

The Apache Qpid Broker-J heavy relies on java reflection mechanism. A static instrumentation agent can be used to replace `method.invoke()` reflection calls with static final `MethodHandle.invokeExact()`.

To use instrumentation agent following JVM argument should be added to the broker start parameters:

```
-javaagent:$BROKER_DIR/lib/qpid-broker-instrumentation-${broker-version}.jar
```

List of classes to instrument can be supplied as a comma separated list:

```
-javaagent:$BROKER_DIR/lib/qpid-broker-instrumentation-${broker-version}.jar=Confi
```

```
-javaagent:$BROKER_DIR/lib/qpid-broker-instrumentation-${broker-version}.jar=Confi
```

```
-javaagent:$BROKER_DIR/lib/qpid-broker-instrumentation-${broker-version}.jar=Confi
```

When no arguments supplied, all classes will be instrumented.

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. High Availability Overview

The Broker provides a HA implementation offering an **Active/Passive** mode of operation. When using HA, many instances of the 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 the Apache Qpid JMS and Apache Qpid JMS AMQP 0-x clients), 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.

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

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

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

Note

The HA solution from the Apache Qpid Broker-J is incompatible with the HA solution offered by the CPP Broker. It is not possible to co-locate Qpid Broker-J 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 or the initial configuration (Appendix H, *BDB HA initial configuration* illustrates how to use initial configuration for BDB HA group creation). 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

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

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.

Figure 10.4. View of group from one node

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*. Flushing behavior can be influenced by virtual host context parameters "qpid.broker.bdbCommiterNotifyThreshold" (defines threshold for amount of messages triggering BDB log flush to the disk) and "qpid.broker.bdbCommiterWaitTimeout" (defines timeout for BDB log flush to the disk).

It is possible to assign a one policy to the master and a different policy to the replicas. These are configured as attributes *localTransactionSynchronizationPolicy* and *remoteTransactionSynchronizationPolicy* 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.

A new master is elected among nodes with the most current set of log files. When there is a tie, the priority is used as a tie-breaker to select amongst these nodes.

The node priority is set as an integer value. A priority of zero is used to ensure that a node cannot be elected master, even if it has the most current set of files.

For convenience, the Web Management Console uses user friendly names for the priority integer values in range from 0 to 3 inclusive. The following priority 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. The priority value for option *Highest* is *3*.
- *High*. Nodes with this priority will be favoured but not as much so as those with Highest. The priority value for this option is *2*.
- *Normal*. This is a default election priority. The priority value for this option is *1*.
- *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. The priority value for this option is *0*.

Node priority is configured as an attribute *priority* 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 *quorumOverride* 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. Allow to Operate Solo

This attribute only applies to groups containing exactly two nodes.

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 `allow to operate solo` flag allows a node in a two node group to elect itself master and to operate sole. It is configured as an attribute *designatedPrimary* 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 `allow to operate solo` flag to it. It will elect itself master and work can continue, albeit from one node.

Important

It is imperative not to allow the `allow to operate solo` flag 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.

10.4.6. Maximum message size

Internally, BDB JE HA restricts the maximum size of replication stream records passed from the master to the replica(s). This helps prevent DOS attacks. If expected application maximum message size is greater than 5MB, the BDB JE setting `je.rep.maxMessageSize` and Qpid context variable `qpid.max_message_size` needs to be adjusted to reflect this in order to avoid running into the BDB HA JE limit.

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 Apache Qpid JMS and Apache Qpid JMS AMQP 0-x clients 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. 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`.

10.8. 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.

10.9. 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.

10.10. Backups

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

10.11. 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-7.4.5.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

BDB module includes the "hot" backup utility `org.apache.qpid.server.store.berkeleydb.BDBBackup`. This utility can perform the backup when the broker is running.

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

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

`java` -cp *`"${QPID_HOME}/lib/*"`* org.apache.qpid.server.store.berkeleydb.BDBBackup \
-fromdir *`${QPID_WORK}/<nodename>/config`* -todir *`path/to/backup/folder`*

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

11.2.2. BDB-HA

See Section 11.2.1, “BDB”

Note

BDB-HA VirtualHost node backups are node specific. You cannot use the backup of one node to recover a different node. To backup a group a backup of each node needs to be taken separately.

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\\9.2.0` (Windows) or `/usr/local/qpid/qpid-broker/9.2.0` (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:+UseG1GC`	This is the preferred mechanism for customising garbage collection behaviour. The value should contain valid garbage collection options(s) for the target JVM. Refer to the JVM's documentation for details.

Environment variable	Default	Purpose
QPID_JAVA_MEM	`-Xmx512m -XX:MaxDirectMemorySize=1536m`	This is the preferred mechanism for customising the size of the JVM's heap and direct memory. The value should contain valid memory option(s) for the target JVM. Oracle JVMs understand `-Xmx` to specify a maximum heap size, `-Xms` an initial size, and `-XX:MaxDirectMemorySize` for the maximum amount of direct memory. For example, `QPID_JAVA_MEM="-Xmx6g -XX:MaxDirectMemorySize=12g"` would set a maximum heap size of 6GB and 12GB of direct memory. 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_status_updates	true	If set true, the Broker will produce operational logging messages.
qpid.broker_disabled_features	none	Allows optional Broker features to be disabled. Currently understood feature names are: `qpid.jms-selector` Feature names should be comma separated.

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 : Destination : AMQP(127.
```

`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 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-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-1010	Platform : JVM : *vendor* version: *version* OS : *operating system vendor* version: *operating system version*} arch: *processor architecture* cores: *number of CPU cores*
	Key information about the environment hosting the Broker
BRK-1011	Maximum Memory : Heap : *size* bytes Direct : *bytes* size
	Configured memory paramters for the Broker.
BRK-1012	Management Mode : User Details : *management node user id* / *management mode password*

Message Id	Message Text / Purpose
	Used when Broker is started in management mode to indicate the management credentials that may be used connect to the Broker.
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.
BRK-1018	Operation : *operation name*
	Indicates that the named operation has been invoked

Table C.4. Management Log Messages

Message Id	Message Text / Purpose
MNG-1001	*type* Management Startup
	Indicates that a Management plugin is starting up. Supported by Web management plugin.
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
	Indicates that a virtualhost has been closed. This occurs on Broker shutdown.

Message Id	Message Text / Purpose
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.
VHT-1008	Operation : *operation name*
	Indicates that the named operation has been invoked

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 in bytes*, Capacity : *resumesize in bytes*, Messages : *size in messages*, Message Capacity : *resumesize in messages*
	Indicates that a queue has exceeded its permitted capacity when *Producer Flow Control* overflow policy is used. See Section 4.7.5, “Controlling Queue Size” for details.
QUE-1004	Underfull : Size : *size in bytes*, Capacity : *resumesize in bytes*, Messages : *size in messages*, Message Capacity : *resumesize in messages*
	Indicates that a queue has fallen to its resume capacity when *Producer Flow Control* overflow policy is used. See Section 4.7.5, “Controlling Queue Size” for details.
QUE-1005	Dropped : *number* messages, Depth : *size* bytes, *size* messages, Capacity : *limit* bytes, *limit* messages
	Indicates that a given number of messages is deleted when *Ring* overflow policy is used and any of queue capacity limits is breached . See Section 4.7.5, “Controlling Queue Size” for details.
QUE-1016	Operation : *operation name*
	Indicates that the named operation has been invoked

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.
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.
EXH-1004	Operation : *operation name*
	Indicates that the named operation has been invoked

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 : Destination : *target port* : Protocol Version : *protocol version* : Client ID : *clientid* : 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.
CON-1004	Connection dropped

Message Id	Message Text / Purpose
	Indicates that a connection has been unexpectedly closed by the peer. This usually occurs if a machine hosting an application fails or the application's process is abruptly terminated.
CON-1005	Client version *version* logged by validation
	Indicates that a connection has been received from client with a version number that is configured to be logged. This feature may help teams manage software currency.
CON-1006	Client version *version* rejected by validation
	Indicates that a connection attempt has been received from client with a version number that is configured to be rejected. This feature may help manage software currency.
CON-1007	Connection close initiated by operator
	Indicates that a connection has been closed by the actions of an Operator using manangement.
CON-1009	Uncommitted transaction(s) contains *size* bytes of incoming message data exceeding *size* bytes limit. Messages will be flowed to disk.
	Warns that the transactions associated with this connection contain so much uncommitted data that a threshold has been breached. The connection responds by flowing the messages already associated with the transactions and any new messages to disk. The connection reverts to normal behaviour once the quantity of uncommitted data falls beneath the threshold. Normally this happens when the transactions commit or rollback.
CON-1010	Open Transaction : *time* ms
	Indicates that a messaging transaction has been open for longer than that permitted. See Section 9.3, “Transaction Timeout” for more details.
CON-1011	Idle Transaction : *time* ms
	Indicates that a messaging transaction has been idle for longer than that permitted. See Section 9.3, “Transaction Timeout” for more details.

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
	Indicates that a channel has been closed.

Message Id	Message Text / Purpose
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 4.7.5, “Controlling Queue Size” for more details.
CHN-1006	Flow Control Removed
	Indicates that producer flow control has been removed from a channel. See Section 4.7.5, “Controlling Queue Size” 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
CHN-1014	Operation : *operation name*
	Indicates that the named operation has been invoked

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.

Message Id	Message Text / Purpose
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 blocked in application code). It may also indicate a generally overloaded system.
SUB-1004	Operation : *operation name*
	Indicates that the named operation has been invoked

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
	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, “Disk quota-based flow control” for details.
MST-1009	Store overfull condition cleared

Message Id	Message Text / Purpose
	The store size has fallen beneath its resume capacity and therefore flow control has been rescinded. See Section 9.2.1, “Disk quota-based 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.
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*
	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*

Message Id	Message Text / Purpose
	This node has been designated primary and can now operate solo. Applies to two node groups only. See Section 10.4.5, “Allow to Operate Solo”
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 *peer* rejected. Maximum connection count (*limit*) for this port already reached.
	Connection from given host is rejected because of reaching the maximum allowed limit
PRT-1007	Unsupported protocol header received *header bytes*, replying with *AMQP version*
	Ususualy indicates an attempt to make an non-AMQP connection on an AMQP port, for instance, with a web browser.
PRT-1008	Connection from *address* rejected
	Incoming connection is rejected because the port's connection limits are already reached.
PRT-1009	FAILED to bind *name* service to *port number*
	The given port number could not be bound because it is already in-use.

Message Id	Message Text / Purpose
PRT-1010	Operation : *operation name*
	Indicates that the named operation has been invoked

Table C.17. Resource Limit Log Messages

Message Id	Message Text / Purpose
RL-1001	Accept
	Connection has been accepted.
RL-1002	Reject
	Connection has been rejected.
RL-1003	Info
	An informational message regarding connection limits.

Table C.18. User Log Messages

Message Id	Message Text / Purpose
USR-1001	Create
	User has been created.
USR-1002	Update
	User has been updated.
USR-1003	Delete
	User has been deleted.

Appendix D. Statistics Reporting

The Broker has the ability to periodically write statistics of any entity within the system to the log. Statistics reporting can be configured for a single entity (e.g. a queue) or for all entities of a particular category (e.g. for all queues). The system can be configured dynamically at runtime without the need for the system to be restarted.

This feature helps allow the behaviour of the overall system to be understood and can aid real-time problem diagnosis.

It can be configured Broker-wide or separately for each virtual host.

The format of the statistics report is configurable.

D.1. Statistics Report Period

This governs the period with which statistics reports will be written to the log. The period is defined in seconds. By default the statistics report period is zero, meaning the system is disabled. To enable the statistics report set the *statistics reporting period* on either the Broker or virtualhost to a non-zero value.

Once the period is defined, the system will respond to the statistic report patterns defined described next.

D.2. Statistic Report Patterns

The statistic report pattern defines the format of a statistic report written to the log.

Statistic report patterns are defined by content variables. The place where the context variable is defined governs the scope i.e. the entities to which the pattern will be applied.

For instance, to define a statistics reporting pattern for a single queue, set the contextvariable on the queue itself. If you want the same statistics report pattern for apply to all queues, set the pattern on a suitable ancestor of the queue. For instance, if set on virtualhost, the pattern will applied to all queues defined on that virtualhost. If set on Broker, the pattern will be applied to all queues on the Broker.

The context variable name is formed as follows: `qpid.<category-name>.statisticsReportPattern`.

For instance, for queue: `qpid.queue.statisticsReportPattern` and virtualhost: `qpid.virtualhost.statisticsReportPattern`

The value of the context variable is a free text string containing reference(s) to the statistic names that are to appear in the report. References are made by surrounding the name of the statistic with '$' and curly braces, thus `${<statistic-name>}`.

Statistics references allow an optional formatters. The supported formatters are: `:byteunit` (produces a human readable byte value e.g. 3 MiB), `:duration` (produces a ISO-8601 duration)and `:datetime` (produces a ISO-8601 date/time).

For example, a statistic report pattern for the `queue` category specifying two queue statistic values: `queueDepthMessages=${queueDepthMessages},queueDepthBytes=${queueDepthBytes:byteunit}`

Like all context variables, the statistic report pattern can also reference the attributes of the entity or even its ancestors. This feature can be exploited to include things like the name of the entity within the report.

These points are illustrated in the examples in the next section.

A catalogue of statistics names and descriptions is available from the REST API documentation available through the Web Management Console.

D.3. Examples

Adding a statistic reporting pattern to a single queue, called `myqueue` using the REST API and cURL. This example uses `ancestor` references to include entity names:

Example D.1. Enabling statistics for a single queue using the REST API and cURL

```
curl --user admin --data '{"name" : "qpid.queue.statisticsReportPattern",
"value" : "${ancestor:virtualhost:name}/${ancestor:queue:name}: queueDepthMessages
queueDepthBytes=${queueDepthBytes:byteunit}"}' https://localhost:8080/api/latest/q
```

Once enabled, an example statistic report output written to the log might look like this:

```
INFO  [virtualhost-default-pool-0] (q.s.Queue)- Statistics: default/myqueue: queue
 INFO  [virtualhost-default-pool-2] (q.s.Queue)- Statistics: default/myqueue: queu
 INFO  [virtualhost-default-pool-2] (q.s.Queue)- Statistics: default/myqueue: queu
```

Removing a statistic report pattern from the same queue:

Example D.2. Disabling statistics for a single queue using the REST API and cURL

```
curl --user admin --data '{"name" : "qpid.queue.statisticsReportPattern"}' https:/
```

Adding a statistic reporting pattern to all queues:

Example D.3. Enabling statistics for all queues using the REST API and cURL

```
curl --user admin --data '{"name" : "qpid.queue.statisticsReportPattern",
"value" : "${ancestor:virtualhost:name}/${ancestor:queue:name}:
oldestMessageAge=${oldestMessageAge:duration}"}' https://localhost:8080/api/latest
```

Once enabled, an example statistic report for a virtualhost with two queues might look like this:

```
INFO  [virtualhost-default-pool-1] (q.s.Queue)- Statistics: default/myqueue1: olde
      INFO  [virtualhost-default-pool-1] (q.s.Queue)- Statistics: default/myqueue2
```

Appendix E. 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
```

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 E.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 F. Miscellaneous

F.1. JVM Installation verification

F.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\jdk-11.0.13\
```

Then confirm that a Java installation (11 or higher) is available:

```
java -version
```

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

```
java version "11.0.13" 2021-10-19 LTS
        Java(TM) SE Runtime Environment 18.9 (build 11.0.13+10-LTS-370)
        Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.13+10-LTS-370, mixed mo
```

F.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/jdk-11.0.13
```

Then confirm that a Java installation (11 or higher) is available:

```
java -version
```

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

```
java version "11.0.13" 2021-10-19 LTS
        Java(TM) SE Runtime Environment 18.9 (build 11.0.13+10-LTS-370)
        Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.13+10-LTS-370, mixed mo
```

F.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` folder.

```
Unix:
cp driver.jar qpid-broker-9.2.0/lib

Windows:
copy driver.jar qpid-broker-9.2.0\lib
```

Appendix G. Queue Declaration Arguments supported by the Broker

Qpid Broker-J supports a number of custom arguments which can be specified as part of *queue.declare* commands for AMQP 0-x protocols. This section provides an overview of the supported arguments.

Table G.1. Queue declare arguments

Argument Name	Description
Declaration of overflow policy. See Section 4.7.5, “Controlling Queue Size” for more details.	
qpid.policy_type	Defines queue overflow policy.
qpid.max_count	Defines *maximum number of messages*.
qpid.max_size	Defines *maximum number of bytes*.
The *Overflow Policy* and the limits can be specified using *Address* based syntax as in the example below: `my-queue; {create: always, node: {x-declare: {arguments: {'qpid.max_count': 10000, 'qpid.max_size': 102400, 'qpid.policy_type: 'ring'}}}}`	
Alternative declaration of *Producer Flow Control* overflow policy. See Section 4.7.5, “Controlling Queue Size” for more details.	
x-qpid-capacity	Defines *maximum number of bytes*.
x-qpid-flow-resume-capacity	Defines flow resume threshold in bytes
The *Producer Flow Control* can be specified using *Address* based syntax as in the example below: `my-queue; {create: always, node: {x-declare: {arguments: {'x-qpid-capacity': 102400, 'x-qpid-flow-resume-capacity': 8192000}}}}`	
x-qpid-priorities	Specifies a priority queue with given number priorities
qpid.queue_sort_key	Specifies sorted queue with given message property used to sort the entries
qpid.last_value_queue_key	Specifies lvq queue with given message property used to conflate the entries
qpid.ensure_nondestructive_consumers	Set to true if the queue should make all consumers attached to it behave non-destructively. (Default is false).

Argument Name	Description
x-qpid-maximum-delivery-count	Specifies this queue's maximum delivery count.
x-single-active-consumer	If set `true`, then of all consumers attached to a queue, only one will be designated as *active*, and eligible to receive messages. If the active consumer is detached, and other consumers are attached, one of these other consumers is selected to become the single active consumer.

Appendix H. BDB HA initial configuration

The section Section 5.1, “Introduction” provides an introduction into Broker-J initial configuration and how broker configuration can be created from initial configuration on first broker start-up. This appendix illustrates how to create a BDB HA group from an initial configuration file. For creation of BDB HA group using Web Management Console please refer Section 6.2, “Web Management Console”.

The BDB HA group usually consists of two or more nodes hosting a distributed virtual host.

When BDB HA node is created the following attributes has to be provided

- *groupName*; a name of BDB HA group
- *nodeName*; a name of BDB HA node
- *address*; a node address as colon-separated pair of host name and port
- *helperAddress*; an address of existing helper node. It is required when node joins an existing group.
- *helperNodeName*; a name of existing helper node. It is required when node joins an existing group.
- *permittedNodes*; an array containing all addresses of nodes allowed to join the group.

A node priority can be optionally specified for the node to influence master election among nodes with the most current set of data. An attribute *priority* is used to specify a priority as an integer number.

Apart from a group name and permitted nodes, the rest of node attribute values varies from node to node.

In order to use the same initial configuration for creation of BDB HA nodes, the context variable can be used for varying attribute values.

In the example of initial configuration illustrated in this apendix, the following context variables are defined.

- *${ha.group_name}*; used to pass an HA group name
- *${ha.node_name}*; used to pass a node name
- *${ha.node_address}*; used to pass a node address
- *${ha.helper_address}*; used to pass an address of helper node.
- *${ha.helper_node_name}*; used to pass an address of helper node.
- *${ha.permitted_nodes}*; used to pass a stringified json array containing permitted nodes for the group.
- *${ha.priority}*; used to pass a node priority.

H.1. Example of BDB HA 'Initial Configuration'

An example of 'Initial Configuration' for BDB HA:

Example H.1. BDB HA 'Initial configuration'

```
{
```

```
"name": "${broker.name}",
"modelVersion" : "9.0",
"authenticationproviders" : [ {
  "name" : "plain",
  "type" : "Plain",
  "users" : [ {
  "name" : "guest",
  "type" : "managed",
  "password" : "guest"
} ]
} ],
"brokerloggers" : [ {
  "name" : "logfile",
  "type" : "File",
  "fileName" : "${qpid.work_dir}${file.separator}log${file.separator}qpid.log",
  "brokerloginclusionrules" : [ {
    "name" : "Root",
    "type" : "NameAndLevel",
    "level" : "WARN",
    "loggerName" : "ROOT"
  }, {
    "name" : "Qpid",
    "type" : "NameAndLevel",
    "level" : "INFO",
    "loggerName" : "org.apache.qpid.*"
  }, {
    "name" : "Operational",
    "type" : "NameAndLevel",
    "level" : "INFO",
    "loggerName" : "qpid.message.*"
  }, {
    "name" : "Statistics",
    "type" : "NameAndLevel",
    "level" : "INFO",
    "loggerName" : "qpid.statistics.*"
  } ]
}, {
  "name" : "memory",
  "type" : "Memory",
  "brokerloginclusionrules" : [ {
    "name" : "Root",
    "type" : "NameAndLevel",
    "level" : "WARN",
    "loggerName" : "ROOT"
  }, {
    "name" : "Qpid",
    "type" : "NameAndLevel",
    "level" : "INFO",
    "loggerName" : "org.apache.qpid.*"
  }, {
    "name" : "Operational",
    "type" : "NameAndLevel",
    "level" : "INFO",
    "loggerName" : "qpid.message.*"
```

```
    }, {
      "name" : "Statistics",
      "type" : "NameAndLevel",
      "level" : "INFO",
      "loggerName" : "qpid.statistics.*"
    } ]
  } ],
  "ports" : [  {
    "name" : "AMQP",
    "port" : "${qpid.amqp_port}",
    "authenticationProvider" : "plain",
    "virtualhostaliases" : [ {
      "name" : "nameAlias",
      "type" : "nameAlias"
    }, {
      "name" : "defaultAlias",
      "type" : "defaultAlias"
    }, {
      "name" : "hostnameAlias",
      "type" : "hostnameAlias"
    } ]
  }, {
  "name" : "HTTP",
    "port" : "${qpid.http_port}",
    "authenticationProvider" : "plain",
    "protocols" : [ "HTTP" ]
  }],
  "virtualhostnodes" : [ {
    "name" : "${ha.node_name}",
    "type" : "BDB_HA",
    "address" : "${ha.node_address}",
    "groupName" : "${ha.group_name}",
    "helperAddress" : "${ha.helper_address}",
    "helperNodeName" : "${ha.helper_node_name}",
    "permittedNodes" : "${ha.permitted_nodes}",
    "priority" : "${ha.priority}",
    "defaultVirtualHostNode" : "true",
    "virtualHostInitialConfiguration" : "${qpid.initial_config_virtualhost_config}
  } ],
  "plugins" : [ {
    "type" : "MANAGEMENT-HTTP",
    "name" : "httpManagement"
  } ]
}
```

H.2. Creation of BDB HA group using an initial configuration.

Let's create a BDB HA group with name `weather` consisting of tree nodes `weather1`, `weather2`, and `weather3` using the initial configuration above. We start node `weather1` on host/port `chaac:5000`, node `weather2` on host/port `indra:5000` and node `weather3` on host/port `thor:5000`.

The node `weather1` can be created with the following command

```
$ ./qpid-server -icp ./initial-config.json -prop ha.node_name=weather1 -prop  ha.n
-prop ha.group_name=weather -prop ha.helper_address=chaac:5000 -prop ha.helper_nod
-prop ha.permitted_nodes="[\"chaac:5000\",\"indra:5000\",\"thor:5000\"]" -prop ha.
-prop qpid.amqp_port=10000 -prop qpid.http_port=20000
```

Please note, the broker is started with initial configuration at file `./initial-config.json`. The context variable for node name `ha.node_name` is set to `weather1`. The node address context variable `ha.node_address` is set to `chaac:5000`. As it is a first node, the helper address is set to the same address as a node address (`ha.helper_address=chaac:5000`) and the helper node name is to itself (`ha.helper_node_name=weather1`). The group name is set to `weather` with `ha.group_name=weather`. The group nodes are listed in `ha.permitted_nodes`. The amqp and http ports are overridden to 10000 and 20000 accordingly. The node priority is set to `3`.

The node `weather2` can be created on host`indra` with the following command:

```
 $ ./qpid-server -icp ./initial-config.json -prop ha.node_name=weather2 -prop  ha.
-prop ha.group_name=weather -prop ha.helper_address=chaac:5000 -prop ha.helper_nod
-prop ha.permitted_nodes="[\"chaac:5000\",\"indra:5000\",\"thor:5000\"]"  -prop ha
-prop qpid.amqp_port=10001 -prop qpid.http_port=20001
```

The context variable for node name `ha.node_name` is set to `weather2`. The node address context variable `ha.node_address` is set `indra:5000`. The amqp and http ports are overridden to 10001 and 20001 accordingly. The node `weather2` priority is set to `2`. The rest of the context variables have the same values as for node `weather1`. The latter is used as a helper node for creation of `weather2`.

The node `weather3` can be created on host`thor` with the following command:

```
$ ./qpid-server -icp ./initial-config.json -prop ha.node_name=weather3 -prop  ha.n
-prop ha.group_name=weather -prop ha.helper_address=chaac:5000 -prop ha.helper_nod
-prop ha.permitted_nodes="[\"chaac:5000\",\"indra:5000\",\"thor:5000\"]"  -prop ha
-prop qpid.amqp_port=10002 -prop qpid.http_port=20002
```

The context variable for node name `ha.node_name` is set to `weather3`. The node address context variable `ha.node_address` is set `thor:5000`. The amqp and http ports are overridden to 10002 and 20002 accordingly. The node `weather3` priority is set to `1`. The rest of the context variables have the same values as for node `weather1` which is used as a helper node for creation of `weather3`.

Chapter 12. Docker Images

12.1. Building Container Image

To use an official Apache release in your image run the following command from the qpid-docker directory where <QPID_RELEASE_VERSION> is the release version you wish to use (e.g. 9.1.0):

```
cd qpid-docker
```

```
docker-build.sh --release <QPID_RELEASE_VERSION>
```

This will download the Qpid Broker-J release and copy all the files necessary to build the pre-configured Docker image and provide you with additional instructions. Follow these instructions to finish building the image you want based on the provided Docker file or even one of your own.

If you would rather prefer to build the docker image from local Broker-J distribution, build the parent project using the command

```
mvn clean install -DskipTests=true
```

Navigate to the module 'qpid-docker':

```
cd qpid-docker
```

Execute the command:

```
docker-build.sh --local-dist-path <PATH_TO_LOCAL_QPID_DISTRIBUTION
```

This will copy all the files necessary to build the pre-configured Docker image and provide you with additional instructions. Follow these instructions to finish building the image you want based on one of the provided Docker file or even one of your own.

12.2. Running the Container

12.2.1. Container Start

Container can be started using following command:

```
docker run -d -p 5672:5672 -p 8080:8080 --name qpid <IMAGE_NAM
```

or

```
podman run -d -p 5672:5672 -p 8080:8080 -v qpid_volume:/qpid-b
```

There are two ports exposed: 5672 for AMQP connections and 8080 for HTTP connections.

There are following environment variables available when running the container:

Table 12.1. Environment Variables

Environment Variable	Description
JAVA_GC	JVM Garbage Collector parameters, default value "-XX:+UseG1GC"
JAVA_MEM	JVM memory parameters, default value "-Xmx300m -XX:MaxDirectMemorySize=200m"
JAVA_OPTS	Further JVM parameters, default value is an empty string

12.2.2. Container Volume

The image will use the directory /qpid-broker-j/work to hold the configuration and the data of the running broker. To persist the broker configuration and the data outside the container, start container with the volume mapping:

```
docker run -d -p 5672:5672 -p 8080:8080 -v <BROKER_DIRECTOR_ON
```

or

```
podman run -d -p 5672:5672 -p 8080:8080 -v <BROKER_DIRECTOR_ON
```

12.2.3. Stopping the Container

Running container can be stopped using following command:

```
docker stop <CONTAINER_NAME>
```

12.3. Broker Users

Default configuration provides a preconfigured broker user, having read and write access to all broker objects:

- admin (default password 'admin')

Username of the 'admin' user can be overridden be providing the variable QPID_ADMIN_USER on start, and the default password of the 'admin' user can be overridden be providing the variable QPID_ADMIN_PASSWORD on start:

```
docker run -d -p 5672:5672 -p 8080:8080 -v qpid_volume:/qpid-broke
```

Further broker users as well as other broker objects (queues, exchanges, keystores, truststore, ports etc.) can be created via HTTP management interface. Description of the broker REST API can be found in broker book (chapter 6.3).

12.4. Broker Customization

To customize broker before building the container image, its configuration files may be edited to start broker with queues, exchanges, users or other objects.

The file config.json contains definitions of the broker objects and references a file containing definitions of virtualhost objects (exchanges and queues).

It may be helpful first to create broker objects needed via broker web GUI or via REST API, and then investigate the configuration files and copy the appropriate definitions to the configuration files used for container image creation.

An example of the default initial configuration JSON file is provided in broker book (chapter 5.7).

12.4.1. Exchanges

To create exchanges a JSON element "exchanges" should be created containing an array of single exchange definitions:

```
"exchanges" : [ {
    "name" : "amq.direct",
    "type" : "direct"
  }, {
    "name" : "amq.fanout",
    "type" : "fanout"
  }, {
    "name" : "amq.match",
    "type" : "headers"
  }, {
    "name" : "amq.topic",
    "type" : "topic"
  }, {
    "name" : "request.QUEUE1",
    "type" : "topic",
    "durable" : true,
    "durableBindings" : [ {
    "arguments" : { },
    "destination" : "QUEUE1",
    "bindingKey" : "#"
  } ],
  "unroutableMessageBehaviour" : "REJECT"
} ]
```

Information about exchanges, their types and properties can be found in broker book (chapter 4.6).

Please note that each virtualhost pre-declares several exchanges, described in the broker book (chapter 4.6.1).

12.4.2. Queues

To create queue a JSON element "queues" should be created containing an array of single queue definitions:

```
"queues" : [ {
    "name" : "QUEUE1",
    "type" : "standard",
    "durable" : true,
    "maximumQueueDepthBytes" : 6144000,
    "maximumQueueDepthMessages" : 6000,
    "messageDurability" : "ALWAYS",
    "overflowPolicy" : "REJECT"
  }, {
    "name" : "QUEUE2",
    "type" : "standard",
    "durable" : true,
    "maximumQueueDepthBytes" : 6144000,
    "maximumQueueDepthMessages" : 6000,
    "messageDurability" : "ALWAYS",
    "overflowPolicy" : "REJECT"
  } ]
```

Information about queues, their types and properties can be found in broker book (chapter 4.7).

12.4.3. Users

Users can be defined in an authentication provider. Authentication providers are defined on broker level (file config.json).

Information about authentication providers, their types and properties can be found in broker book (chapter 8.1).

Examples for most commonly used authentication providers can be found below.

12.4.3.1. Anonymous Authentication Provider

```
"authenticationproviders" : [ {
    "name" : "anon",
    "type" : "Anonymous"
} ]
```

For additional details see broker book (chapter 8.1.5).

12.4.3.2. Plain Authentication Provider

```
"authenticationproviders" : [{
    "name" : "plain",
    "type" : "Plain",
    "secureOnlyMechanisms" : [],
    "users" : [ {
        "name" : "admin",
        "type" : "managed",
        "password" : "<PASSWORD>"
    } ]
} ]
```

For additional details see broker book (chapter 8.1.7).

12.4.3.3. ACL Rules

The ACL rules for users are defined in file broker.acl following the syntax:

```
ACL {permission} {<group-name>|<user-name>|ALL} {action|AL
```

The predefined broker.acl file contains permissions for the 'admin' user:

```
# account 'admin' - enabled all actions
ACL ALLOW-LOG admin ALL ALL
```

For additional details see broker book (chapter 8.3.2).

12.4.4. Overriding Broker Configuration

Customized configuration for the Broker-J instance can be used by replacing the files residing in the work folder with the custom ones, e.g. config.json or default.json. Put the replacement files inside a folder and map it as a volume to:

```
docker run -d -p 5672:5672 -p 8080:8080 -v <DIRECTORY_ON_HOST>:/qpid-broker-j/
```

The contents of work-override folder will be copied over to work folder first time after the instance creation so that the broker will start with user-supplied configuration.