Red Hat Enterprise MRG 2 Messaging Installation and Configuration Guide

Installing and Configuring the Red Hat Enterprise Messaging Server

David Ryan Joshua Wulf

Red Hat Enterprise MRG 2 Messaging Installation and Configuration Guide

Installing and Configuring the Red Hat Enterprise Messaging Server

David Ryan

Joshua Wulf
jwulf@redhat.com

Keywords

Abstract
This guide covers the installation and configuration of the Red Hat Enterprise Messaging Server.

Table of Contents

Preface

1. Document Conventions

1.1. Default Programming Language for Code Samples

If this book contains programming samples in more than one programming language, you can set your preferred programming language here.

C#/.NET C++ Java JavaScript Node.js Python Ruby

When code samples are available in multiple languages, your language will be presented by default.

2. We Appreciate Your Feedback

Each section in this book has an small link at the end, on the right hand side of the page: "Something wrong with this?". Click this link to give us feedback.

New and Updated Content

1. Changed for 2.3

The following content is changed for the 2.3 release of the documentation:

See Also:

- Section 4.2, “Create and Configure Queues using qpid-config”
- Section 8.2.6, “Enable SSL in Python Clients”
- Section 9.3.2.3, “Configure Queue Replication on the Source Broker”
- Section 12.1.3, “Using qpid-config”
- Appendix A, *Exchange and Queue Declaration Arguments*

Report a bug

2. New for 2.3

The following content is new for the 2.3 release of the documentation.

See Also:

- Section 5.3.4.5, “Messaging Broker Memory Requirements”
- Chapter 7, *Logging*
- Section 7.2, “Change Broker Logging Verbosity”
- Section 7.3, “Tracking Object Lifecycles”
- Section 8.1.2, “SASL Support in Windows Clients”
- Section 8.1.3, “SASL Mechanisms”
- Section 8.3.2, “Default ACL File”
- Section 8.3.8, “Enforcing Queue Size Limits via ACL”
- Section 8.3.9, “Resource Quota Options”
- Section 8.3.10, “Routing Key Wildcards”
- Section 8.3.11, “User Name and Domain Name Symbol Substitution”
- Section 8.2.3, “Export an SSL Certificate for Clients”
- Section 11.6, “Deploying the Qpid JCA Adapter on JBoss EAP 6”
- Section 12.1.2, “Changes in qpid-config and qpid-stat for MRG 2.3”
- Appendix B, *OpenSSL Certificate Reference*

Report a bug

Chapter 1. Messaging Components - Installation

1.1. The Messaging Server

1.1.1. The Messaging Server

The MRG Messaging Server is an Enterprise-grade tested and supported messaging server based on the Apache Qpid project.

See Also:

- Section 5.3.4.5, “Messaging Broker Memory Requirements”

Report a bug

1.1.2. Messaging Broker

Red Hat Enterprise Messaging uses the Apache Qpid C++ broker to store, forward, and distribute messages.

Report a bug

1.1.3. Install the Messaging Server on Red Hat Enterprise Linux 5

1. Subscribe your system to the `Red Hat MRG Messaging` channel in Red Hat Network.
2. Install the MRG Messaging group using the yum command:

```
yum groupinstall "MRG Messaging"
```

Report a bug

1.1.4. Install the Messaging Server on Red Hat Enterprise Linux 6

1. If you are using RHN classic management for your system, in addition to the base channel for Red Hat Enterprise Linux 6, subscribe your system to the following channel:
   - `Additional Services Channels for Red Hat Enterprise Linux 6 / MRG Messaging v.2 (for RHEL-6 Server)`
   - `Release Channels for Red Hat Enterprise Linux 6 / RHEL Server High Availability`
2. Install the MRG Messaging group using the yum command (as root):

```
yum groupinstall "Messaging Client Support"
yum groupinstall "MRG Messaging"
yum install qpid-cpp-server-cluster
```

Report a bug

1.1.5. Firewall Configuration

You must allow incoming connections on the port used by the Messaging System. The default port for AMQP traffic is 5672.

On a Red Hat Enterprise Linux system, the firewall is provided by `iptables`. Commands modifying the firewall configuration must be run with root privileges. The following sequence of commands will configure `iptables` to allow incoming connections on port 5672.

```
iptables -I INPUT -p tcp -m tcp --dport 5672  -j ACCEPT
service iptables save
service iptables restart
```

To view the currently configured firewall rules, issue the command:

```
service iptables status
```

Report a bug

Chapter 2. Starting the Broker

2.1. Starting the Broker via command line vs as a service

When started as a service, the Broker reads its start-up options from a configuration file. When started at the command line, the Broker can read its start-up options either from a configuration file, or from command-line arguments.

Starting the Broker as a service is useful for production servers. The Broker can be configured to automatically start whenever the server is restarted. For development use, starting and re-starting the Broker with different configurations usually means that starting from the command line is useful.

Report a bug

2.2. Running the Broker at the command line

2.2.1. Start the Broker at the command line

Start the Broker

1. By default, the broker is installed in `/usr/sbin/`. If this is not on your path, you need to type the whole path to start the broker:

```
/usr/sbin/qpidd -t
```

You will see output similar to the following when the broker starts:

```
[date] [time] info Loaded Module: libbdbstore.so.0
[date] [time] info Locked data directory: /var/lib/qpidd
[date] [time] info Management enabled
[date] [time] info Listening on port 5672
```

The `-t` or `--trace` option enables debug tracing, printing messages to the terminal.

Note: The locked data directory `/var/lib/qpidd` is used for persistence, which is enabled by default.

Report a bug

2.2.2. Stop the Broker when started at the command line

1. To stop the broker, type **`CTRL`**+ **`C`** at the shell prompt

```
[date] [time] notice Shutting down.
[date] [time] info Unlocked data directory: /var/lib/qpidd
```

Report a bug

2.3. Running the Broker as a service

2.3.1. Run the Broker as a service

- For production use, Red Hat Enterprise Messaging is usually run as a service. To start the broker as a service, with root privileges run the command:

```
service qpidd start
```

The message broker will start with the message:

```
Starting Qpid AMQP daemon:                                 [  OK  ]
```

Report a bug

2.3.2. Stop the Broker service

- Check the status of a broker running as a service with the `service status` command. Stop the broker with the `service stop`.

```
# service qpidd status
qpidd (pid PID) is running...

# service qpidd stop
Stopping Qpid AMQP daemon:                                 [  OK  ]
```

Report a bug

2.3.3. Configure the Broker service to start automatically when the server is started

Typically on a production server you wish the message broker to start automatically when the machine is restarted. To do this, you need to enable the `qpidd` service.

- Run the following command as root:

```
chkconfig qpidd on
```

The message broker qpidd demon will now start automatically when the server is started.

Report a bug

2.4. Run multiple Brokers on one machine

2.4.1. Running multiple Brokers

In a development environment you might wish to run multiple brokers on the same machine for testing or prototyping.

It is possible.

Report a bug

2.4.2. Start multiple Brokers

To run more than one broker on a single machine, they must run on different ports and use different directories for the journals.

1. Select two available ports, for example *5555* and *5556*.
2. Start each new broker, using the `--data-dir` command to specify a new data directory for each:

```
$ qpidd -p 5555 --data-dir /tmp/qpid/store/1
```

```
$ qpidd -p 5556 --data-dir /tmp/qpid/store/2
```

Report a bug

Chapter 3. Broker options

3.1. Set Broker options at the command line

To set options for a single instance, add the option to the command line when you start the broker.

- This example uses the command line option `-t` to start the broker with debug tracing.

```
$ /usr/sbin/qpidd -t
```

Command-line options need to be provided each time the broker is invoked from the command line.

Report a bug

3.2. Set Broker options in a configuration file

To set the Broker options when running the Broker as a service, or to create a set of options that can be used when starting the Broker from the command-line, but without having to specify them each time on the command-line, use a configuration file.

1. Become the root user, and open the `/etc/qpidd.conf` file in a text editor.
2. This example uses the configuration file to enable debug tracing. Changes will take effect from the next time the broker is started and will be used in every subsequent session.

```
# Configuration file for qpidd
trace=1
```

3. If you are running the broker as a service, you need to restart the service to reload the configuration options.

```
# service qpidd restart
Stopping qpidd daemon:                     [  OK  ]
Starting qpidd daemon:                     [  OK  ]
```

4. If you are running the broker from the command-line, start the broker with no command-line options to use the configuration file.

```
# /usr/sbin/qpidd
[date] [time] info Locked data directory: /var/lib/qpidd
[date] [time] info Management enabled
[date] [time] info Listening on port 5672
```

Report a bug

3.3. Broker options

3.3.1. Options for running the Broker as a Daemon

Table 3.1. Options for running the broker as a service (daemon)

Options for running the broker as a daemon	
`-d`	Run in the background as a daemon. Log messages from the broker are sent to `syslog` (`/var/log/messages`) by default.
`-q`	Shut down the broker that is currently running.
`-c`	Check if the daemon is already running. If it is running, return the process ID.
`-d --wait=<seconds>`	Set the maximum wait time (in seconds) for the daemon to initialize. If the daemon has not successfully completed initialization within this time, an error is returned. This option must be used in conjunction with the `-d` option, or it will be ignored.

Report a bug

3.3.2. General Broker options

Table 3.2. General Broker Options

General options for running the broker	
`-t`	This option enables verbose log messages, for debugging only.
`-p <Port_Number>`	Instructs the broker to use the specified port. Defaults to port 5672. It is possible to run multiple brokers simultaneously by using different port numbers.
`-v`	Displays the installed version.
`-h`	Displays the help message.

Report a bug

3.3.3. Logging

By default, log output is sent to `stderr` if the broker is run on the command line, or to `syslog` (`/var/log/messages/`), if the broker is run as a service.

Table 3.3. Logging Options

Options for logging with `syslog`	
`-t [--trace]`	Enables all logging
`--log-enable RULE (notice+)`	Enables logging for selected levels and components. RULE is in the form `LEVEL[+][:PATTERN]`. Levels are one of: `trace, debug, info, notice, warning, error, critical`.. For example: `--log-enable warning+` logs all warning, error, and critical messages. `--log-enable debug:framing` logs debug messages from the framing namespace. This can be used multiple times.
`--log-time yes|no`	Include time in log messages
`--log-level yes|no`	Include severity level in log messages
`--log-source`	Include source file:line in log messages
`--log-thread yes|no`	Include thread ID in messages
`--log-function yes|no`	Include function signature in log messages
`--log-hires-timestamp yes|no (0)`	Use hi-resolution timestamps in log messages
`--log-category yes|no (1)`	Include category in log messages
`--log-prefix STRING`	Prefix to append to all log messages
`--log-to-stderr yes|no`	Send logging output to `stderr`. Enabled by default when run from command line.
`--log-to-stdout yes|no`	Send logging output to `stdout`.
`--log-to-file` *`FILE`*	Send log output to the specified filename. *`FILE`*.
`--log-to-syslog yes|no`	Send logging output to `syslog`. Enabled by default when run as a service.
`--syslog-name` *`NAME`*	Specify the name to use in `syslog` messages. The default is `qpidd`.
`--syslog-facility` *`LOG_XXX`*	Specify the facility to use in `syslog` messages. The default is `LOG_DAEMON`.

See Also:

- Chapter 7, *Logging*

Report a bug

3.3.4. Modules

Table 3.4. Options for using modules with the broker

Options for using modules with the broker	
`--load-module` *`MODULENAME`*	Use the specified module as a plug-in.
`--module-dir <DIRECTORY>`	Use a different module directory.
`--no-module-dir`	Ignore module directories.

Getting Help with Modules

To see the help text for modules, use the `--help` command:

```
# /usr/sbin/qpidd --help
```

Report a bug

3.3.5. Persistence Options

Table 3.5. Persistence Options

Persistence Options	
`--store-dir` *DIRECTORY*	Specify a directory for journals and persistent configuration. The default is `/var/lib/qpidd` when run as a daemon, or `~/.qpidd` when run from the command line.
`--num-jfiles` *NUMBER*	The number of files for each instance of the persistence journal. The default is 8. Minimum is 4; maximum is 64. The total size in bytes of each journal is `num-jfiles` * `jfile-size-pgs` * 64 * 1024.
`--jfile-size-pgs` *NUMBER*	The size of each journal file, in multiples of 64KB. The default is 24. Minimum is 1; maximum is 32767. The total size of each journal in bytes is `num-jfiles` * `jfile-size-pgs` * 64. The default size for a journal is 1.5 megabytes. The minimum size is 64 kilobytes, the maximum size is 2 gigabytes.
`--wcache-page-size` *NUMBER*	\ The size (in KB) of the pages in the write page cache. Allowable values must be powers of 2 (1, 2, 4, ... 128). Lower values decrease latency but also decrease throughput. The default is 32.
`--tpl-num-jfiles` *NUMBER*	The number of files for each instance of the TPL journal. The default is 8. Minimum is 4; maximum is 64.
`--tpl-jfile-size-pgs` *NUMBER*	The size of each TPL journal file in multiples of 64KB. The default is 24. Minimum is 1; maximum is 32767.
`--tpl-wcache-page-size` *NUMBER*	The size (in KB) of the pages in the TPL write page cache. Allowable values must be powers of 2 (1, 2, 4, ... 128). Lower values decrease latency but also decrease throughput. The default is 32.
`--truncate yes|no`	If yes, truncates the store, discarding any existing records. If no, preserves any existing stores for recovery. The default is no.
`--no-data-dir`	Disables storage of configuration information and other data. If the default directory at `/var/lib/qpidd` exists, it will be ignored.

Illegal Numeric Parameters

When out-of-range or illegal parameters are supplied to the store, they are replaced with the closest legal value, and a warning is reported in the log file. This applies to all numeric store parameters.
For example, starting the store with `--num-jfiles 1` (the minimum value is 4), the store automatically substitutes a value of 4 and places the following warning into the log file:
`"warning parameter num-jfiles (1) is below allowable minimum (4); changing this parameter to minimum value."`
Similarly, if a value which is not a power of 2 is given for the `wcache-page-size` parameter, the closest power of 2 will be substituted with a warning in the log file.

Report a bug

3.3.6. Resource Quota Options

Changes

The maximum number of connections can be restricted:

Table 3.6. Resource Quota Options

Option	Description	Default Value
`--max-connections` *N*	total concurrent connections to the broker	500

Notes

- `--max-connections` is a qpid core limit and is enforced whether ACL is enabled or not.
- `--max-connections` is enforced per Broker. In a cluster of *N* nodes where all Brokers set the maximum connections to 20 the total number of allowed connections for the cluster will be *N**20.

ACL-based Quotas

To enable ACL-based quotas, an ACL file must be loaded:

Table 3.7. ACL Command-line Option

Option	Description	Default Value
`--acl-file` *FILE (policy.acl)*	The policy file to load from, loaded from data dir	

When an ACL file is loaded, the following ACL options can be specified at the command-line to enforce resource quotas:

Table 3.8. ACL-based Resource Quota Options

Option	Description	Default Value
`--connection-limit-per-user` *N*	`MRG 2.2+` The maximum number of connections allowed per user. 0 implies no limit.	0
`--connection-limit-per-ip` *N*	`MRG 2.2+` The maximum number of connections allowed per host IP address. 0 implies no limit.	0
`--max-queues-per-user` *N*	`MRG 2.3+` Total concurrent queues created by individual user	0

Notes

- In a cluster system the actual number of connections may exceed the connection quota value `N` by one less than the number of member nodes in the cluster. For example: in a 5-node cluster, with a limit of 20 connections, the actual number of connections can reach 24 before limiting takes place.
- Cluster connections are checked against the connection limit when they are established. The cluster connection is denied if a free connection is not available. After establishment, however, a cluster connection does not consume a connection.
- Allowed values for `N` are 0..65535.
- These limits are enforced per *cluster*.
- A value of zero (0) disables that option's limit checking.
- Per-user connections are identified by the authenticated user name.
- Per-ip connections are identified by the `<broker-ip><broker-port>-<client-ip><client-port>` tuple which is also the management connection index.
  - With this scheme host systems may be identified by several names such as `localhost` IPv4, `127.0.0.1` IPv4, or `::1` IPv6, and a separate set of connections is allowed for each name.
  - Per-IP connections are counted regardless of the user credentials provided with the connections. An individual user may be allowed 20 connections but if the client host has a 5 connection limit then that user may connect from that system only 5 times.

Chapter 4. Queues

4.1. Message Queue

Message Queues are the mechanism for consuming applications to subscribe to messages that are of interest.

Queues receive messages from exchanges, and buffer those messages until they are consumed by message consumers. Those message consumers can browse the queue, or can acquire messages from the queue. Messages can be returned to the queue for redelivery, or they can be rejected by a consumer.

Multiple consumers can share a queue, or a queue may be exclusive to a single consumer.

Message producers can create and bind a queue to an exchange and make it available for consumers, or they can send to an exchange and leave it up to consumers to create queues and bind them to the exchange to receive messages of interest.

Temporary private message queues can be created and used as a response channel. Message queues can be set to be deleted by the broker when the application using them disconnects. They can be configured to group messages, to update messages in the queue with newly-arriving copies of messages, and to prioritise certain messages.

Report a bug

4.2. Create and Configure Queues using qpid-config

Changes

The `qpid-config` command line tool can be used to create and configure queues.

The complete command reference for `qpid-config` is available by running the command with the `--help`switch:

```
qpid-config --help
```

When no server is specified, `qpid-config` runs against the message broker on the current machine. To interact with a message broker on another machine, use the `-a` or `--broker-addr` switch. For example:

```
qpid-config -a server2.testing.domain.com
```

The argument for the broker address option can specify a username, password, and port as well:

```
qpid-config -a user1/secretpassword@server2.testing.domain.com:5772
```

To create a queue, use the `qpid-config add queue` command. This command takes the name for the new queue as an argument, and [optionally] queue options.

A simple example, creating a queue called `testqueue1` on the message broker running on the local machine:

```
qpid-config add queue testqueue1
```

Here are the various options that you can specify when creating a queue with `qpid-config`:

Table 4.1. Options for qpid-config add queue

Options for qpid-config add queues	
`--passive, --dry-run`	*Removed in MRG 2.3.* Don't create the queue, just check that the command syntax is correct.
`--alternate-exchange` *queue name*	Name of the alternate exchange. When the queue is deleted, all remaining messages in this queue are routed to this exchange. Messages rejected by a queue subscriber are also sent to the alternate exchange.
`--durable`	The new queue is durable. It will be recreated if the server is restarted, along with any undelivered messages marked as PERSISTENT sent to this queue.
`--cluster-durable`	The new queue becomes durable if there is only one functioning node in a cluster.
`--file-count` *integer*	The number of files in the queue's persistence journal.
`--file-size` *integer*	File size in pages (64KiB/page).
`--max-queue-size` *integer*	Maximum in-memory queue size as bytes. Note that on 32-bit systems queues will not go over 3GB, regardless of the declared size.
`--max-queue-count` *integer*	Maximum in-memory queue size as a number of messages.
`--limit-policy` *[none, reject, flow-to-disk, ring, ring-strict]*	Action to take when queue limit is reached.
`--order` *[fifo, lvq, lvq-no-browse]*	*Removed in MRG 2.3*. Queue ordering policy. This method of declaring Last Value Queues is deprecated. See the updated information on Last Value Queues for instructions on declaring LVQs using arguments.
`--generate-queue-events` *integer*	If set to `1`, every enqueue will generate an event that can be processed by registered listeners (e.g. for replication). If set to 2, events will be generated for both enqueues and dequeues.
`--flow-stop-size` *integer*	Turn on sender flow control when the number of queued bytes exceeds this value.
`--flow-resume-size` *integer*	Turn off sender flow control when the number of queued bytes drops below this value.
`--flow-stop-count` *integer*	Turn on sender flow control when the number of queued messages exceeds this value.
`--flow-resume-count`	Turn off sender flow control when the number of queued messages drops below this value.
`--group-header`	Enable message groups. Specify name of header that holds group identifier.
`--shared-groups`	Allow message group consumption across multiple consumers.
`--argument` *name=value*	Specify a key-value pair to add to the queue arguments. This can be used, for example, to specify `no-local=true` to suppress loopback delivery of self-generated messages.

Note that you cannot create an exclusive queue using `qpid-config`, as an exclusive queue is only available in the session where it is created.

See Also:

- Section 12.1.2, “Changes in qpid-config and qpid-stat for MRG 2.3”
- Section 12.1.3, “Using qpid-config”
- Section 4.6.2, “Declaring a Last Value Queue”
- Section 4.5, “Ignore Locally Published Messages”
- Appendix A, *Exchange and Queue Declaration Arguments*

Report a bug

4.3. Memory Allocation Limit (32-bit)

A Broker running on a 32-bit operating system has a 3GB memory allocation limit. You can create a queue with greater than 3GB capacity on such a system, however when the queue reaches 3GB of queued messages, an attempt to send more messages to the queue will result in a memory allocation failure.

Report a bug

4.4. Exclusive Queues

Exclusive queues can only be used in one session at a time. When a queue is declared with the exclusive property set, that queue is not available for use in any other session until the session that declared the queue has been closed.

If the server receives a declare, bind, delete or subscribe request for a queue that has been declared as exclusive, an exception will be raised and the requesting session will be ended.

Note that a session close is not detected immediately. If clients enable heartbeats, then session closes will be determined within a guaranteed time. See the client APIs for details on how to set heartbeats in a given API.

Report a bug

4.5. Ignore Locally Published Messages

You can configure a queue to discard all messages published using the same connection as the session that owns the queue. This suppresses a message loop-back when an application publishes messages to an exchange that it is also subscribed to.

To configure a queue to ignore locally published messages, use the `no-local` key in the queue declaration as a key:value pair. The value of the key is ignored; the presence of the key is sufficient.

For example, to create a queue that discards locally published messages using `qpid-config`:

```
qpid-config add queue noloopbackqueue1 --argument no-local=true
```

Note that multiple distinct sessions can share the same connection. A queue set to ignore locally published messages will ignore all messages from the *connection* that declared the queue, so all sessions using that connection are local in this context.

Report a bug

4.6. Last Value (LV) Queues

4.6.1. Last Value Queues

Last Value Queues allow messages in the queue to be overwritten with updated versions. Messages sent to a Last Value Queue use a header key to identify themselves as a version of a message. New messages with a matching key value arriving on the queue cause any earlier message with that key to be discarded. The result is that message consumers who browse the queue receive the latest version of a message only.

Report a bug

4.6.2. Declaring a Last Value Queue

Last Value Queues are created by supplying a `qpid.last_value_queue_key` when creating the queue.

For example, to create a last value queue called 'stock-ticker' that uses 'stock-symbol' as the key, using `qpid-config`:

```
qpid-config add queue stock-ticker --argument qpid.last_value_queue_key=stock-
symbol
```

To create the same queue in an application:

Python

```
myLastValueQueue = mySession.sender("stock-ticker;{create:always,
node:{type:queue, x-declare:{arguments:{'qpid.last_value_queue_key': 'stock-
symbol'}}}}")
```

Report a bug

4.7. Message Groups

4.7.1. Message Groups

Message Groups allow a sender to indicate that a group of messages should all be handled by the same consumer. The sender sets the header of messages to identify them as part of the same group, then sends the messages to a queue that has message grouping enabled.

The broker ensures that a single consumer gets exclusive access to the messages in a group, and that the messages in a group are delivered and re-delivered in the order they were received.

Note that Message Grouping cannot be used in conjunction with Last Value Queue or Priority Queuing.

The implementation of Message Groups is described in a specification attached to its feature request: QPID-3346: Support message grouping with strict sequence consumption across multiple consumers.

Report a bug

4.7.2. Message Group Consumer Requirements

The correct handling of group messages is the responsibility of both the broker and the consumer. When a consumer fetches a message that is part of a group, the broker makes that consumer the owner of that message group. All of the messages in that group will be visible only to that consumer *until the consumer acknowledges receipt of all the messages it has fetched from that group*. When the consumer acknowledges all the messages it has fetched from the group, the broker releases its ownership of the group.

The consumer should acknowledge all of the fetched messages in the group at once. The purpose of message grouping is to ensure that all the messages in the group are dealt with by the same consumer. If a consumer takes grouped messages from the queue, acknowledges some of them and then disconnects due to a failure, the unacknowledged messages in the group will be released and become available to other consumers. However, the acknowledged messages in the group have been removed from the queue, so now part of the group is available on the queue with the header `redelivered=True`, and the rest of the group is missing.

For this reason, consuming applications should be careful to acknowledge all grouped messages at once.

Report a bug

4.7.3. Configure a Queue for Message Groups using qpid-config

This example `qpid-config` command creates a queue called "MyMsgQueue", with message grouping enabled and using the header key "GROUP_KEY" to identify message groups.

```
qpid-config add queue MyMsgQueue --group-header="GROUP_KEY" --shared-groups
```

Report a bug

4.7.4. Create a Queue with Message Groups enabled

To create a queue with message groups enabled, specify values for `qpid.group_header_key` and `qpid.shared_msg_group` in the queue creation arguments.

The `qpid.group_header_key` is the header key that will be used to match messages on. Messages with the same value for this key in their header belong to the same group.

`qpid.shared_msg_group` should be set to `1`.

The following example creates an auto-deleting queue that uses the header field "msgGroupID" to group messages:

Python

```
groupedSender = session.sender("my-grouped-msg-queue; {create: always, node:
{x-declare: {auto-delete: True, arguments: {'qpid.group_header_key':
'msgGroupID', 'qpid.shared_msg_group': 1}}}}")
```

C++

```
groupedSender = session.createSender("my-grouped-msg-queue; {create:always,
node: {x-declare: {auto-delete: True, arguments:
{'qpid.group_header_key':'msgGroupID', 'qpid.shared_msg_group':1}}}}")
```

Report a bug

4.7.5. Message Groups Demonstration

The following Python program demonstrates the use and behavior of message groups. To run this program, copy and paste the code into a text file and save it as `message-groups.py`, then run it using Python on a machine with the messaging broker started.

The program creates an auto-deleting queue with messaging enabled or disabled, then sends messages to the queue with a message group header that matches the group header for the queue. When messaging is enabled it demonstrates how consumers are given ownership of a message group by the broker, and how this affects what they see and do not see on the queue. It also demonstrates how consumers release ownership of a group by acknowledging all the messages they have fetched from that group, and how group ownership is not released by partially acknowledging the fetched messages.

The program uses two different connections to simulate two consumers, who would usually be running as separate processes, perhaps on different machines.

Python

```
import sys
from qpid.messaging import *

def sendmsg(group, num):
# send the message to the broker and add it to our in-memory representation of
the broker queue
  global memoryqueue
  global tx

  msg = Message(group + num)
  msg.properties = {'ourGroupID': group}

  tx.send(msg)
  memoryqueue.append(group + num)

def pullmsg(consumer):
# fetch a message from the broker and print it to the console
  global counter
  global memoryqueue

  msg = consumers[consumer - 1].fetch(timeout = 1)

  print "\nQueued message: " + memoryqueue[counter]
  print "Consumer " + str(consumer) + " got: " + msg.content

  counter +=1
  return msg

# Two connections are used to simulate two distinct consumers
connection = Connection("localhost:5672")
connection2 = Connection("localhost:5672")
connection.open()
connection2.open()

try:
  session = connection.session()
  session2 = connection2.session()

  x = raw_input('Enable message grouping [Y/n]?')

  if x == 'N' or x == 'n':

    # Create the queue without message groups
    tx = session.sender("test-nogroup-queue; {create: always, node:{x-
declare:{auto-delete:True}}}")
    rx1 = session.receiver("test-nogroup-queue")
    rx2 = session2.receiver("test-nogroup-queue")

    print "\nMessage grouping is disabled"
    msggroup = False

  else:

    # Create the queue with message groups enabled
    tx = session.sender("test-group-queue; {create: always, node:{x-
declare:{auto-delete: True, arguments: {'qpid.group_header_key': 'ourGroupID',
'qpid.shared_msg_group' : 1}}}}")
    rx1 = session.receiver("test-group-queue")
    rx2 = session2.receiver("test-group-queue")

    print "\nMessage grouping is enabled"
    msggroup = True

# Put the receivers in an array so we can use a function to fetch messages
```

```
  consumers = []
  consumers.append(rx1)
  consumers.append(rx2)

  print "Sending interleaved messages from two different groups to the
queue..."

# We create an in-memory picture of the queue, to see what order the messages
are on the broker
  memoryqueue = []

  sendmsg('A', '1')
  sendmsg('B', '1')
  sendmsg('B', '2')
  sendmsg('A', '2')
  sendmsg('B', '3')
  sendmsg('A', '3')

  counter = 0
  pullmsg(1)
  pullmsg(2)

  if msggroup:
    print "\nConsumer 1 now owns message group A. Consumer 2 now owns message
group B."

  msgc1 = pullmsg(1)
  msgc2 = pullmsg(2)

  if msggroup:
    print "\nThe consumers will now acknowledge all the messages, or only the
last one."
    resp = raw_input('Should they acknowlege all messages? [Y/n]')

    if resp == 'N' or resp == 'n':
      print "\nAcknowledging only part of the group. The consumers retain
ownership of the group. This is an anti-pattern! See the source code comments
for details."

      session.acknowledge(msgc1)
      session2.acknowledge(msgc2)
      antipattern = True

      # Acknowledging only part of a group is an anti-pattern. Messages are
grouped to ensure that a single consumer can deal with the whole group. If this
consumer now fails before completing the rest of the group, the unacknowledged
messages in the group will be released and redelivered by the broker, but the
acknowledged messages in the group are now missing in action!

    else:
      print "\nAcknowledging all fetched messages. The consumers will release
ownership of the groups."
      session.acknowledge()
      session2.acknowledge()
      antipattern = False

    print "\nPulling more messages from the queue:"

  pullmsg(1)
  pullmsg(2)
  if msggroup:
    if antipattern == False:
      print "\nConsumer 1 now owns message group B. Consumer 2 now owns
message group A."
```

```
    print "\nSending some more messages to the queue..."

  sendmsg('B', '4')
  sendmsg('B', '5')
  sendmsg('A', '4')
  sendmsg('A', '5')

  pullmsg(1)
  pullmsg(2)
  pullmsg(1)
  pullmsg(2)

finally:
  connection.close()
  connection2.close()
```

Example program output

The program sends messages from two different Groups - `A` and `B` - to a queue. Here is an example of the output when message groups are disabled:

```
$ python message-groups.py
Enable message grouping [Y/n]?n

Message grouping is disabled
Sending interleaved messages from two different groups to the queue...

Queued message: A1
Consumer 1 got: A1

Queued message: B1
Consumer 2 got: B1

Queued message: B2
Consumer 1 got: B2

Queued message: A2
Consumer 2 got: A2

Queued message: B3
Consumer 1 got: B3

Queued message: A3
Consumer 2 got: A3

Queued message: B4
Consumer 1 got: B4

Queued message: B5
Consumer 2 got: B5

Queued message: A4
Consumer 1 got: A4

Queued message: A5
Consumer 2 got: A5
```

The consumers are pulling messages from the queue in a round-robin fashion, and they see the messages on the queue in the order the messages were sent there.

Running the program with message groups enabled demonstrates how message groups influence how consumers see the messages on the queue:

```
$ python message-groups.py
Enable message grouping [Y/n]?y

Message grouping is enabled
Sending interleaved messages from two different groups to the queue...

Queued message: A1
Consumer 1 got: A1

Queued message: B1
Consumer 2 got: B1

Consumer 1 now owns message group A. Consumer 2 now owns message group B.

Queued message: B2
Consumer 1 got: A2

Queued message: A2
Consumer 2 got: B2
```

At this point of the program you can choose to acknowledge all of the acquired messages, or only some of them. Acknowledging all of the messages that have been acquired so far releases ownership of the group, and the next messages that the consumers see will be the next messages on the queue:

```
The consumers will now acknowledge all the messages, or only the last one.
Should they acknowlege all messages? [Y/n]y

Acknowledging all fetched messages. The consumers will release ownership of the
groups.

Pulling more messages from the queue:

Queued message: B3
Consumer 1 got: B3

Queued message: A3
Consumer 2 got: A3
```

They will then take ownership of the groups of those messages:

```
Consumer 1 now owns message group B. Consumer 2 now owns message group A.

Sending some more messages to the queue...

Queued message: B4
Consumer 1 got: B4

Queued message: B5
Consumer 2 got: A4

Queued message: A4
Consumer 1 got: B5

Queued message: A5
Consumer 2 got: A5
```

If you instead choose to acknowledge only the last message, rather than all the acquired messages in the group, then the program will warn you that this is an anti-pattern, and demonstrate that the consumers retain ownership of the group:

```
The consumers will now acknowledge all the messages, or only the last one.
Should they acknowlege all messages? [Y/n]n

Acknowledging only part of the group. The consumers retain ownership of the group.
This is an anti-pattern! See the source code comments for details.

Pulling more messages from the queue:

Queued message: B3
Consumer 1 got: A3

Queued message: A3
Consumer 2 got: B3

Sending some more messages to the queue...

Queued message: B4
Consumer 1 got: A4

Queued message: B5
Consumer 2 got: B4

Queued message: A4
Consumer 1 got: A5

Queued message: A5
Consumer 2 got: B5
```

Report a bug

4.7.6. Default Group

All messages arriving to a queue with message groups enabled with no group identifier in their header are considered to belong to the same "default" group. This group is `qpid.no-group`. If a message cannot be assigned to any other group, it is assigned to this group.

Report a bug

4.7.7. Override the Default Group Name

When a queue has message groups enabled, messages are grouped based on a match with a header field. Messages that have no match in their headers for a group are assigned to the default group. The default group is preconfigured as `qpid.no-group`. You can change this default group name by supplying a value for the `default-message-group` configuration parameter to the broker at start-up. For example, using the command line:

```
qpidd --default-message-group "EMPTY-GROUP"
```

Report a bug

4.8. Alternate Exchanges

4.8.1. Rejected and Orphaned Messages

Messages can be explicitly *rejected* by a consumer. When a message is fetched over a reliable link, the consumer must acknowledge the message for the broker to release it. Instead of acknowledging a message, the consumer can *reject* the message. The broker discards rejected messages, unless an alternate exchange has been specified for the queue, in which case the

broker routes rejected messages to the alternate exchange.

Messages are orphaned when they are in a queue that is deleted. Orphaned messages are discarded, unless an alternate exchange is configured for the queue, in which case they are routed to the alternate exchange.

Report a bug

4.8.2. Alternate Exchange

An *alternate exchange* provides a delivery alternative for messages that cannot be delivered via their initial routing.

For an alternate exchange specified for a queue, two types of unroutable messages are sent to the alternate exchange:

1. Messages that are acquired and then rejected by a message consumer (*rejected messages*).
2. Unacknowledged messages in a queue that is deleted (*orphaned messages*).

For an alternate exchange specified for an exchange, one type of unroutable messages is sent to the alternate exchange:

1. Messages sent to the exchange with a routing key for which there is no matching binding on the exchange.

Note that a message will not be re-routed a second time to an alternate exchange if it is orphaned or rejected after previously being routed to an alternate exchange. This prevents the possibility of an infinite loop of re-routing.

However, if a message is routed to an alternate exchange and is unable to be delivered by that exchange because there is no matching binding, then it *will* be re-routed to that exchange's alternate exchange, if one is configured. This ensures that fail-over to a dead letter queue is possible.

Report a bug

4.9. Queue Sizing

4.9.1. Controlling Queue Size

Controlling the size of queues is an important part of performance management in a messaging system.

When queues are created, you can specify a maximum queue size (`qpid.max_size`) and maximum message count (`qpid.max_count`) for the queue.

`qpid.max_size` is specified in bytes. `qpid.max_count` is specified as the number of messages.

The following `qpid-config` creates a queue with a maximum size in memory of 200MB, and a maximum number of 5000 messages:

```
qpid-config add queue my-queue --max-queue-size=204800000 --max-queue-count 5000
```

In an application, the `qpid.max_count` and `qpid.max_size` directives go inside the `arguments` of the `x-declare` of the node. For example, the following address will create the queue as the `qpid-config` command above:

Python

```
tx = ssn.sender("my-queue; {create: always, node: {x-declare: {'auto-delete':
True, arguments:{'qpid.max_count': 5000, 'qpid.max_size': 204800000}}}}")
```

Note that the `qpid.max_count` attribute will only be applied if the queue does not exist when this code is executed.

Behavior when limits are reached: `qpid.policy_type`

The behavior when a queue reaches these limits is configurable. By default, on non-`durable` queues the behavior is `reject`: further attempts to send to the queue result in a `TargetCapacityExceeded` exception being thrown at the sender.

The configurable behavior is set using the `qpid.policy_type` option. The possible values are:

reject
Message publishers throw an exception `TargetCapacityExceeded`. This is the default behavior for non-`durable` queues.

flow-to-disk
Content of messages that exceed the limit are removed from memory and held on disk. Header and other information needed to track the message state on the queue is retained in memory. This policy makes sense when the message body is significant larger than the headers. Note that the messages stored to disk are not persistent unless the queue is a `durable` queue and the message is marked persistent.

ring
The oldest messages are removed to make room for newer messages.

ring-strict
Similar to the ring policy, but will not remove messages that have not yet been accepted by a client. If the limit is exceeded and the oldest message has not been accepted, the publisher will receive an exception.

The following example `qpid-config` command sets the limit policy to `ring-strict`:

```
qpid-config add queue my-queue --max-queue-size=204800 --max-queue-count 5000 --
limit-policy ring-strict
```

The same thing is achieved in an application like so:

Python

```
tx = ssn.sender("my-queue; {create: always, node: {x-declare: {'auto-delete':
True, arguments:{'qpid.max_count': 5000, 'qpid.max_size': 204800,
'qpid.policy_type: 'ring-strict'}}}}")
```

See Also:

- Section 4.11, “Producer Flow Control”

Report a bug

4.9.2. Enforcing Queue Size Limits via ACL

In MRG 2.3 and above, the maximum queue size can be enforced via an ACL. This allows the administrator to disallow users from creating queues that could consume too many system resources.

CREATE QUEUE rules have ACL rules that limit the upper and lower bounds of both in-memory queue and on-disk queue store sizes.

Table 4.2. Queue Size ACL Rules

User Option	ACL Limit Property	Units
qpid.max_size	queuemaxsizelowerlimit	bytes
	queuemaxsizeupperlimit	bytes
qpid.max_count	queuemaxcountlowerlimit	messages
	queuemaxcountupperlimit	messages
qpid.file_size	filemaxsizelowerlimit	pages (64Kb per page)
	filemaxsizeupperlimit	pages (64Kb per page)
qpid.file_count	filemaxcountlowerlimit	files
	filemaxcountupperlimit	files

ACL Limit Properties are evaluated when the user presents one of the options in a CREATE QUEUE request. If the user's option is not within the limit properties for an ACL Rule that would allow the request, then the rule is matched with a Deny result.

Limit properties are ignored for Deny rules.

Example:

```
# Example of ACL specifying queue size constraints
# Note: for legibility this acl line has been split into multiple lines.
acl allow bob@QPID create queue name=q6 queuemaxsizelowerlimit=500000
                                        queuemaxsizeupperlimit=1000000
                                        queuemaxcountlowerlimit=200
                                        queuemaxcountupperlimit=300
```

These limits come into play when a queue is created as illustrated here:

```
int main(int argc, char** argv) {
const char* url = argc>1 ? argv[1] : "amqp:tcp:127.0.0.1:5672";
const char* address = argc>2 ? argv[2] :
    "message_queue; “
    “ { create: always, “
    “   node: “
    “   { type: queue, “
    “     x-declare: ”
    “     { arguments: “
    “       { qpid.max_count:101,”
    “         qpid.max_size:1000000”
    “       }”
    “     }”
    “   }”
    “ }";
std::string connectionOptions = argc > 3 ? argv[3] : "";

Connection connection(url, connectionOptions);
try {
    connection.open();
    Session session = connection.createSession();
    Sender sender = session.createSender(address);
...
```

This queue can also be created with the `qpid-config` command:

```
qpid-config add queue --max-queue-size=1000000 --max-queue-count=101
```

When the ACL rule is processed assume that the actor, action, object, and object name all match and so this allow rule matches for the allow or deny decision. However, the ACL rule is further constrained to limit 500000 <= max_size <= 1000000 and 200 <= max_count <= 300. Since the `queue_option max_count` is 101 then the size limit is violated (it is too low) and the allow rule is returned with a deny decision.

Note that it is not mandatory to set *both* an upper limit *and* a lower limit. It is possible to set only a lower limit, or only an upper limit.

Report a bug

4.9.3. Queue Threshold Alerts

Queue Threshold Alerts are issued by the broker when a queue with a capacity limit set (either `qpid.max_size` or `qpid.max_count`) approaches 80% of its limit. The figure of 80% is configurable across the server using the broker option `--default-event-threshold-ratio`. If you set this to zero, alerts are disabled for all queues by default. Additionally, you can override the default alert threshold per-queue using `qpid.alert_count` and `qpid.alert_size` when creating the queue.

The Alerts are sent via the QMF framework. You can subscribe to the alert messages by listening to the address `qmf.default.topic/agent.ind.event.org_apache_qpid_broker.queueThresholdExceeded.#`. Alerts are sent as map messages.

The following code demonstrates subscribing to and consuming alert messages:

Python

```
conn = Connection.establish("localhost:5672")
session = conn.session()
rcv =
session.receiver("qmf.default.topic/agent.ind.event.org_apache_qpid_broker.queu
eThresholdExceeded.#")
while True:
  event = rcv.fetch()
  print "Threshold exceeded on queue %s" %
event.content[0]["_values"]["qName"]
  print "     at a depth of %s messages, %s bytes" %
(event.content[0]["_values"]["msgDepth"],
event.content[0]["_values"]["byteDepth"])
  session.acknowledge()
```

Alert Repeat Gap

To avoid alert message flooding, there is a 60 second gap between alert messages. This can be overridden on a per-queue basis using the `qpid.alert_repeat_gap` to specify a different value in seconds.

Backwards-compatible aliases

The following aliases are maintained for compatibility with earlier clients:

- `x-qpid-maximum-message-count` is equivalent to `qpid.alert_count`
- `x-qpid-maximum-message-size` is equivalent to `qpid.alert_size`
- `x-qpid-minimum-alert-repeat-gap` is equivalent to `qpid.alert_repeat_gap`

Report a bug

4.10. Deleting Queues

4.10.1. Delete a Queue with qpid-config

The following `qpid-config` command deletes an empty queue:

```
qpid-config del queue queue-name
```

The command will check that the queue is empty before performing the delete, and will report an error and not delete the queue if the queue contains messages.

To delete a queue that contains messages, use the `--force` switch:

```
qpid-config del queue queue-name --force
```

Report a bug

4.10.2. Automatically Deleted Queues

Queues can be configured to *auto-delete*. The broker will delete an auto-delete queue when it has no more subscribers, or if it is auto-delete *and* exclusive, when the declaring session ends.

Applications can delete queues themselves, but if an application fails or loses its connection it may not get the opportunity to clean up its queues. Specifying a queue as auto-delete delegates the responsibility to the broker to clean up the queue when it is no longer needed.

Auto-deleted queues are generally created by an application to *receive* messages, for example: a response queue to specify in the "reply-to" property of a message when requesting information from a service. In this scenario, an application creates a queue for its own use and subscribes it to an exchange. When the consuming application shuts down, the queue is deleted automatically. The queues created by the `qpid-config` utility to receive information from the message broker are an example of this pattern.

A queue configured to `auto-delete` is deleted by the broker after the last consumer has released its subscription to the queue. After the `auto-delete` queue is created, it becomes eligible for deletion as soon as a consumer subscribes to the queue. When the number of consumers subscribed to the queue reaches zero, the queue is deleted.

Here is an example using the Python API to create an auto-delete queue with the name "my-response-queue":

Python

```
responsequeue = session.receiver('my-response-queue; {create:always, node:{x-
declare:{auto-delete:True}}}')
```

Note: since no bindings are specified in this queue creation, it will be bound to the server's `default` exchange, a pre-configured nameless direct exchange.

A timeout can be configured to provide a grace period before the deletion occurs. If a timeout of 120 seconds is specified, for example, then the broker will wait for 120 seconds after the last consumer disconnects from the queue before deleting it. If a consumer subscribes to the queue within that grace period, the queue is not deleted. This is useful to allow for a consumer to drop its connection and reconnect without losing the information in its queue.

Here is an example using the Python API to create an auto-delete queue with the name "my-response-queue" and an auto-delete timeout of 120 seconds:

Python

```
responsequeue = session.receiver("my-response-queue; {create:always, node:{x-
declare:{auto-delete:True, arguments:{'qpid.auto_delete_timeout':120}}}}")
```

Be aware that a public auto-deleted queue can be deleted while your application is still sending to it, if your application is not holding it open with a receiver. You will not receive an error because you are sending to an exchange, which continues to exist; however your messages will not go to the now non-existent queue. If you are publishing to a self-created auto-deleted queue you should consider carefully if you should be using an auto-deleted queue. If the answer is "yes" (it can be useful for tests that clean up after themselves), then subscribe to the queue when you create it. Your subscription will then act as a handle, and the queue will not be deleted until you release it. Using the Python API:

Python

```
testqueue = session.sender("my-test-queue; {create:always, node:{x-
declare:{auto-delete:True}}}")
testqueuehandle = session.receiver("my-test-queue")
    .....
connection.close()
# testqueuehandle is now released
```

An exception to the requirement that a consumer subscribe and then unsubscribe to invoke the auto-deletion is a queue configured to be `exclusive` and `auto-delete`; these queues are

deleted by the broker when the session that declared the queue ends, since the session that declared the queue is only possible subscriber.

Report a bug

4.10.3. Queue Deletion Checks

When a queue deletion is requested, the following checks occur:

- If ACL is enabled, the broker will check that the user who initiated the deletion has permission to do so.
- If the `ifEmpty` flag is passed the broker will raise an exception if the queue is not empty
- If the `ifUnused` flag is passed the broker will raise an exception if the queue has subscribers
- If the queue is exclusive the broker will check that the user who initiated the deletion owns the queue

Report a bug

4.11. Producer Flow Control

4.11.1. Flow Control

The broker implements producer flow control on queues that have limits set. This blocks message producers that risk overflowing a destination queue. The queue will become unblocked when enough messages are delivered and acknowledged.

Flow control relies on a reliable link between the sender and the broker. It works by holding off acknowledging sent messages, causing message producers to reach their sender replay buffer capacity and stop sending.

Queues that have been configured with a Limit Policy of type `ring` or `ring-strict` do *not* have queue flow thresholds enabled. These queues deal with reaching capacity through the `ring` and `ring-strict` mechanisms. All other queues with limits have two threshold values that are set by the broker when the queue is created:

flow_stop_threshold
the queue resource utilization level that enables flow control when exceeded. Once crossed, the queue is considered in danger of overflow, and the broker will cease acknowledging sent messages to induce producer flow control. Note that *either* queue size or message count capacity utilization can trigger this.

flow_resume_threshold
the queue resource utilization level that disables flow control when dropped below. Once crossed, the queue is no longer considered in danger of overflow, and the broker again acknowledges sent messages. Note that once trigger by either, *both* queue size and message count must fall below this threshold before producer flow control is deactivated.

The values for these two parameters are percentages of the capacity limits. For example, if a queue has a `qpid.max_size` of 204800 (200MB), and a `flow_stop_threshold` of `80`, then the broker will initiate producer flow control if the queue reaches 80% of 204800, or 163840 bytes of enqueued messages.

When the resource utilization of the queue falls below the `flow_resume_threshold`, producer flow control is stopped. Setting the `flow_resume_threshold` above the `flow_stop_threshold`

has the obvious consequence of locking producer flow control on, so don't do it.

Report a bug

4.11.2. Queue Flow State

The flow control state of a queue can be determined by the `flowState` boolean in the queue's QMF management object. When this is `true` flow control is active.

The queue's management object also contains a counter `flowStoppedCount` that increments each time flow control becomes active for the queue.

Report a bug

4.11.3. Broker Default Flow Thresholds

The default flow Control Thresholds can be set for the broker using the following two broker options:

- `--default-flow-stop-threshold` = flow control activated at this percentage of capacity (size or count)
- `--default-flow-resume-threshold` = flow control de-activated at this percentage of capacity (size or count)

For example, the following command starts the broker with flow control set to activate by default at 90% of queue capacity, and deactivate when the queue drops back to 75% capacity:

```
qpidd --default-flow-stop-threshold=90 --default-flow-resume-threshold=75
```

Report a bug

4.11.4. Disable Broker-wide Default Flow Thresholds

To turn off flow control on all queues on the broker by default, start the broker with the default flow control parameters set to 100%:

```
qpidd --default-flow-stop-threshold=100 --default-flow-resume-threshold=100
```

Report a bug

4.11.5. Per-Queue Flow Thresholds

You can set specific flow thresholds for a queue using the following arguments:

`qpid.flow_stop_size`
`integer` flow stop threshold value in bytes.

`qpid.flow_resume_size`
`integer` flow resume threshold value in bytes.

`qpid.flow_stop_count`
`integer` flow stop threshold value as a message count.

`qpid.flow_resume_count`
`integer` flow resume threshold value as a message count.

To disable flow control for a specific queue, set the flow control parameters for that queue to zero.

Report a bug

Chapter 5. Persistence

5.1. Persistent Messages

A persistent message is a message that must not be lost, even if the broker fails.

When a message is marked as persistent *and* sent to a durable queue, it will be written to disk, and resent on restart if the broker fails or shutdowns.

Messages marked as persistent and sent to non-durable queues will not be persisted by the broker.

Note that messages sent using the JMS API are marked persistent by default. If you are sending a message using the JMS API to a durable queue, and do not wish to incur the overhead of persistence, set the message persistence to false.

Messages sent using the C++ API are not persistent by default. To mark a message persistent when using the C++ API, use `Message.setDurable(true)` to mark a message as persistent.

Report a bug

5.2. Durable Queues and Guaranteed Delivery

5.2.1. Configure persistence stores

The Red Hat Enterprise Messaging broker enables persistence by default. Persistence is implemented in the `msgstore.so` module. To verify that persistence is active, make sure that the log shows that the journal is created and the store module initialized when the broker is started. The broker log will contain a line:

```
notice Journal "TplStore": Created
```

Important

If the persistence module is not loaded, messages and the broker state will not be stored to disk, even if the queue is marked durable, and messages are marked persistent.

The `--store-dir` command specifies the directory used for the persistence store and any configuration information. The default directory is `/var/lib/qpidd` when `qpidd` is run as a service, or `~/.qpidd` when `qpidd` is run from the command line. If `--store-dir` is not specified, a subdirectory named `rhm` is created within the directory identified by `--data-dir`; if `--store-dir` is not specified, and `--no-data-dir` is specified, an error is raised.

Important

Only one running broker can access a data directory at a time. If another broker attempts to access the data directory it will fail with an error stating: *`Exception: Data directory is locked by another process.`*

Report a bug

5.2.2. Durable Queues

By default, the lifetime of a queue is bound to the execution of the server process. When the server shuts down the queues are destroyed, and need to be re-created when the broker is restarted. A *durable queue* is a queue that is automatically re-established after a broker is restarted due to a planned or unplanned shutdown.

When the server shuts down and the queues are destroyed, any messages in those queues are lost. As well as automatic re-creation on server restart, durable queues provide *message persistence* for messages that request it. Messages that are marked as persistent and sent to a durable queue are stored and re-delivered when the durable queue is re-established after a shutdown.

Note that not all messages sent to a durable queue are persistent - only those that are marked as persistent. Note also that marking a message as persistent has no effect if it is sent to a queue that is non-durable. A message must be marked as persistent and sent to a durable queue to be persistent.

Report a bug

5.2.3. Create a durable queue using qpid-config

Use the `--durable` option with `qpid-config add queue` to create a durable queue. For example:

```
qpid-config add queue --durable durablequeue
```

Report a bug

5.2.4. Cluster Durable Queues

A queue declared as `cluster durable` will act as a normal, non-durable queue when the other nodes in the cluster are running. It will not provide persistence for messages that are marked as persistent. Clusters provide fault-tolerance and high-availability. In a cluster, reliability of message delivery is provided by the redundancy of cluster nodes. Durable queues have a storage and performance overhead. They provide a persistence store for messages marked as persistent. When the cluster is running, that functionality is redundant, and the performance overhead of durable queues is unwarranted.

However, when the other nodes in a cluster fail, the persistence store of a durable queue becomes the only failsafe for reliable delivery. At that point, incurring the performance and storage overhead of a durable queue makes sense. When a cluster node detects that it is the only remaining node in the cluster, all queues that were declared as `cluster durable` will act as durable queues, and provide message persistence and recovery for messages that are marked as persistent.

Report a bug

5.2.5. Create a cluster durable queue using qpid-config

To create a cluster durable queue using `qpid-config`, use the `--cluster-durable` option:

```
qpid-config add queue clusterdurablequeue1 --cluster-durable
```

See Also:

- Section 4.2, “Create and Configure Queues using qpid-config”

Report a bug

5.2.6. Mark a message as persistent

A *persistent message* is a message that must not be lost even if the broker fails. To make a message persistent, set the delivery mode to `PERSISTENT`. For instance, in C++, the following code makes a message persistent:

```
message.getDeliveryProperties().setDeliveryMode(PERSISTENT);
```

If a persistent message is delivered to a durable queue, it is written to disk when it is placed on the queue.

When a message producer sends a persistent message to an exchange, the broker routes it to any durable queues, and waits for the message to be written to the persistent store, before acknowledging delivery to the message producer. At this point, the durable queue has assumed responsibility for the message, and can ensure that it is not lost even if the broker fails. If a queue is not durable, messages on the queue are not written to disk. If a message is not marked as persistent, it is not written to disk even if it is on a durable queue.

Table 5.1. Persistent Message and Durable Queue Disk States

A persistent message AND durable queue	Written to disk
A persistent message AND non-durable queue	Not written to disk
A non-persistent message AND non-durable queue	Not written to disk
A non-persistent message AND durable queue	Not written to disk

When a message consumer reads a message from a queue, it is not removed from the queue until the consumer acknowledges the message (this is true whether or not the message is persistent or the queue is durable). By acknowledging a message, the consumer takes responsibility for the message, and the queue is no longer responsible for it.

Report a bug

5.2.7. Durable Message State After Restart

When a durable queue is re-established after a restart of the broker, any messages that were marked as persistent and were not reliably delivered before the broker shut down are recovered. The broker does not have information about the delivery status of these messages. They may have been delivered but not acknowledged before the shutdown occurred. To warn receivers that these messages have potentially been previously delivered, the broker sets the `redelivered` flag on *all* recovered persistent messages.

Consuming applications should treat the `redelivered` flag as a suggestion.

Report a bug

5.3. Message Journal

5.3.1. Message Journal

Red Hat Enterprise Messaging allows the size and number of files and caches used for

persistence to be configured. There is one journal for each queue; it records each enqueue, dequeue, or transaction event, in order.

Each journal is implemented as a circular queue on disk, with a read cache and a write cache in memory. On disk, each circular queue consists of a set of files. The caches are page-oriented. When persistent messages are written to a durable queue, the associated events accumulate in the write cache until a page is filled or a timeout occurs, then the page is written to the circular queue using AIO. Messages in the write cache have not yet been acknowledged to the publisher, and can not be read by a consumer until they have been written to the journal. The page size affects performance - smaller page sizes reduce latency, larger page sizes increase throughput by reducing the number of write operations.

The journal files are prepared and formatted when the associated queue is first declared. This doubles throughput with AIO on the first pass, and also guarantees that needed space is allocated. However, this can result in a noticeable delay when durable queues are declared. When file size is increased, the delay is greater.

Report a bug

5.3.2. Configuring the Journal

Journal size is configured per queue, and the configuration can be done programmatically or using the `qpid-config` command line tool.

Report a bug

5.3.3. Configure the Message Journal using qpid-config

When you create a queue using `qpid-config`, you can set the size of the journal using the `--file-count` and `--file-size` options.

Report a bug

5.3.4. Determining Journal Size

5.3.4.1. Preventative design

Applications that use persistent queues must either prevent enqueue threshold exceptions, or respond appropriately when the exception occurs. Prevention can result in better system performance and reliability than exception handling, although both should be implemented.

Matching the rate of publishing with the rate of consuming through distribution is often the best solution. Configuring producer flow control on a persistent queue allows the queue to block producers when the queue is in danger of overflowing.

See Also:

- Section 4.11, “Producer Flow Control”

Report a bug

5.3.4.2. Journal size considerations

Important

Because the size of a circular queue is fixed, it is important to make journals large enough to contain the maximum number of messages anticipated. If the journal becomes approximately 80% full, no new messages can be enqueued, and an attempt to enqueue further messages results in an enqueue threshold exception (`RHM_IORES_ENQCAPTHRESH`). Dequeues are still allowed, and each dequeue frees up space in the store, so enqueues can continue once sufficient space has been freed.

Because transactions can span many persistent queues, enqueue and dequeue events within a transaction are placed in a dedicated persistent queue called the TPL (Transaction Prepared List). When a transaction is committed, the associated events are written to the journal before the transaction commit is acknowledged to the message consumer and messages published in the transaction are made available to message consumers.

If a journal becomes too full to accept new messages, message publishers encounter an enqueue threshold exception (`RHM_IORES_ENQCAPTHRESH`) when the journal is roughly 80% full. Message consumers can still read messages, making room for new messages.

Applications that use persistent queues must either prevent enqueue threshold exceptions, or respond appropriately when the exception occurs. Here are some ways an application might respond if it encounters such exceptions:

- Pause to allow messages to be consumed from the queue, then reconnect and resume sending.
- Publish using a different routing key, or change bindings to route to a different queue.
- Perform load balancing that matches the rate of publishing with the rate of consuming. *Producer flow control* is one mechanism that can be used to achieved this.

Enqueue threshold exceptions will only occur if messages are allowed to accumulate beyond the capacity of a given persistent queue, which usually means that the maximum number of messages on the queue at any given time is large, or the maximum size of these messages is large. In these cases, you may need to increase the size of the journals.

However, increasing the size of the journal has a cost: if the journal is very large, creating a new persistent queue results in a noticeable delay while the journal files are initialized - for journals of multiple megabytes, this delay may be up to several seconds. In some applications, this delay may not be an issue, especially if all persistent queues are created before time-critical messaging begins. In such applications, using very large journals can be a good strategy. If you need to minimize the time needed to create a persistent queue, journals should be large enough to prevent enqueue threshold exceptions, but no larger than necessary.

See Also:

- Section 4.11, “Producer Flow Control”

5.3.4.3. Queue Depth

Queue depth refers to the number of messages on a queue at a given time. No matter how large your journal is, it will eventually fill if messages are published to a persistent queue faster than they are consumed.

To prevent queue threshold exceptions, an application must ensure that messages are consumed and acknowledged, and that messages will not exceed the capacity of the queue journal. The way this is done depends on the application. For instance, some applications may ensure that messages are processed as quickly as possible, keeping queue depth to a

minimum. Other applications may process all messages from a given queue periodically, ensuring that the journal size is large enough to accommodate any messages that might conceivably accumulate in the meantime.

In some applications, the rate at which messages are produced and consumed is known. In other applications, `qpid-tool` can be used to monitor queue depth over time, providing an initial value that can be used to estimate maximum queue depth. The concept body goes here.

Report a bug

5.3.4.4. Estimating Message Size and Queue Size

The maximum journal size needed for a persistent queue depends on queue depth and message size. If your application domain allows you to confidently determine maximum queue depth, message header size, and message content size, you can calculate this size using formulas presented in the following sections. In many cases, it is easier and more reliable to write messaging clients that simulate worst-case scenarios for your application, and use `qpid-tool` to observe minimum, maximum, and average message size and queue depth over time.

To avoid enqueue threshold exceptions, we recommend journal sizes that are double the maximum queue observed in such a simulation.

Report a bug

5.3.4.5. Messaging Broker Memory Requirements

The amount of memory required by a Broker is a function of the number and size of messages it will process simultaneously.

The size of a message is the combination of the size of the message header and the size of the message body.

Calculate message size

Note: Transactions increase the size of messages. Refer to the Journal Size calculations for details of the message size impact of transactions.

Procedure 5.1. Estimate message size

This method allows you to calculate message size theoretically.

1. Default message header content (such as Java timestamp and message-id): 55 bytes
2. Routing Key (for example: a routing key of "testQ" = 5 bytes)
3. Java clients add:
   - `content-type` (for "text/plain" it is 10 bytes)
   - `user-id` (user name passed to SASL for authentication, number of bytes equal to string size)
4. Application headers:
   - Application header overhead: 8 bytes
   - For any textual header properties: `property_name_size + property_value_size + 4` bytes

For example, sending a message using spout such as the following:

```
./run_example.sh org.apache.qpid.example.Spout -c=1 -b="guest:guest@localhost:5672"
-P=property1=value1 -P=property2=value2 "testQ; {create:always}" "123456789"
```

sends an AMQP message with message body size 9, while the message header will be of size:

- 55 bytes for the default size

- 5 bytes for the routing key "testQ"
- 10 bytes for content-type "text/plain"
- 5 bytes for user-id "guest"
- 8 bytes for using application headers
- 9+6+4 bytes for the first property
- 9+6+4 bytes for the second property
- Total header size: 121 bytes
- Total message size: 130 bytes

Procedure 5.2. Determine message size from logs

This method allows you to measure message sizes from a running broker.

1. Enable trace logging by adding the following to `/etc/qpidd.conf`:

   ```
   log-enable=trace+:qpid::SessionState::receiverRecord
   log-enable=info+
   log-to-file=/path/to/file.log
   ```

   Note that this logging will consume significant disk space, and should be turned off by removing these lines after the test is performed.
2. (Re)start the Broker.
3. Send a sample message pattern from a qpid client. This sample message pattern should correspond to your normal utilization, so that the message header and body average sizes match your projected real-world use case.
4. After the message pattern is sent, grep in the logfile for log records:

   ```
   2012-10-16 08:56:20 trace guest@QPID.2fa0df51-6131-463e-90cc-45895bea072c:
   recv cmd 2: header (121 bytes); properties={{MessageProperties: content-
   length=9; message-id=d096f253-56b9-33df-9673-61c55dcba4ae; content-
   type=text/plain; user-id=guest; application-
   headers={property1:V2:6:str16(value1),property2:V2:6:str16(value2)};
   }{DeliveryProperties: priority=4; delivery-mode=2; timestamp=1350370580363;
   exchange=; routing-key=testQ; }}
   ```

   This example log entry contains both header size (121 bytes in this case) and message body size (9 bytes in this case, as content-length=9).

Message memory utilization on Broker

On the broker, memory is utilized to hold the message. In addition:

- A second instance of the message header is kept - one is stored as raw bytes, the other as a map.
- The Message object uses 600 bytes.
- Each message is guarded by three mutexes and monitor. These require 208 bytes.

So the total calculation of memory usage for a message is:

message_body + (message_header * 2) + 808 bytes

Using an average value for message body and header size, and multiplying this figure by the sum of queue depths will give you a saturated memory load figure for the Broker.

Note: an in-memory optimization for exchanges uses one copy of a message for all subscribed queues when the message is delivered to an exchange. This means that a broker with exchanges delivering to multiple queues will use significantly lower amounts of memory in normal operation. However, if the broker is restarted and the queued messages are read from disk, they are read *per-queue* and the full memory impact is experienced.

Report a bug

5.3.4.6. Calculate Journal Size (Without Transactions)

When transactions are not used, the encoded size of a message is the total of the following:

- Enqueue record header (32 bytes, fixed)
- Message header size (Known from problem domain)
- Message size (Known from problem domain)
- Enqueue record tail (12 bytes, fixed)

If the encoded size is not a multiple of 128-bytes, it must be rounded up to the next 128-byte boundary to determine the disk footprint of the message.

> **Example 5.1. Calculate Journal Size (Without Transactions)**
>
> This example shows how to calculate journal size. Here are the characteristics of the messaging queue for this example:
>
> - Average message size: 150 bytes.
> - Maximum queue depth: 25,000 messages at most on disk at any one moment.
> - Message header: 75 bytes average.
> - No transactions, no message sent to multiple queues.
>
> Use these characteristics to calculate the required size:
>
> - An average enqueue record will consume 32 (enqueue record header) + 150 (msg) + 75 (msg header) + 12 (enqueue record tail) = 269 bytes.
> - 269 bytes requires three 128-byte blocks per record, 3 * 128 = 384 bytes.
> - Estimated disk footprint for 25,000 messages = 384 * 25,000 = 9,600,000 bytes ? 9.2 MiB.
> - Double the estimated disk footprint to determine the recommended journal size

Report a bug

5.3.4.7. Impact of Transactions on the Journal

When transactions are used, a transaction ID (XID) is added to each record. The size of the XID is 24 bytes for local transactions. For distributed transactions, the user supplies the XID, which is usually obtained from the transaction monitor, and may be any size. In a transaction, in addition to message enqueue records, journal records are maintained for each message dequeue, and for each transaction abort or commit.

Report a bug

5.3.4.8. Calculate Journal Size (With Transactions)

The following lists provide the encoded size of each of these items. If the encoded size of any item is not a multiple of 128-bytes, it must be rounded up to the next 128-byte boundary to determine the disk footprint of the item. When a transaction is prepared, it is written to disk in 512-byte blocks; since individual records have 128-byte block alignment, empty 128-byte filler records are used to align the write block to 512 bytes if required.

The encoded size of a message enqueue, using transactions, is the total of the following:

Calculate Journal Size (With Transactions)

- Enqueue record header (32 bytes, fixed)
- Transaction ID (XID) size (24 bytes for local transactions, arbitrary size for distributed transactions)
- Message header size (Known from problem domain)
- Message size (Known from problem domain)
- Enqueue record tail (12 bytes, fixed)

The encoded size of a message dequeue, using transactions, is the total of the following:

Message dequeue size (with transactions)

- Dequeue header size (32 bytes, fixed)
- Transaction ID (XID) size (24 bytes for local transactions, arbitrary size for distributed transactions)
- Enqueue record tail (12 bytes, fixed)

The encoded size of a transaction abort or commit is the total of the following:

Message commit/abort size (with transactions)

- Commit / abort header size (32 bytes, fixed)
- Transaction ID (XID) size (24 bytes for local transactions, arbitrary size for distributed transactions)
- Enqueue record tail (12 bytes, fixed)

> **Example 5.2. Calculating Journal Size (With Transactions)**
>
> This example shows how to calculate journal size when distributed transactions are used with a persistent queue. The XID will be supplied by the user in this case. Here are the characteristics of the messaging queue for this example:
>
> - Average message size: 150 bytes, randomly distributed.
> - Maximum queue depth: 25,000 messages at most on disk at any one moment.
> - Enqueues are all transactional using a transaction depth of 1 (ie one enqueue per transaction). There is no more than one open enqueue transaction and one open dequeue transaction at a time.
> - XID size: human-readable UUID format (53 bytes)
> - Message header: 75 bytes average.
>
> Use these characteristics to calculate the required size:
>
> - An average enqueue record consumes 32 (enqueue record header) + 150 (msg) + 53 (XID) + 75 (msg header) + 12 (enqueue record tail) = 322 bytes.
> - 322 bytes requires three 128-byte blocks per record, 3 * 128 = 384 bytes.
> - When the transaction is prepared, this requires one 512-byte page per enqueue record.
> - A transaction commit/abort record is 32 (enqueue record header) + 43(XID) + 12 (enqueue record tail), which requires one 5-12 byte page per abort/commit record.
> - Maximum queue depth is 20,000, so total estimated disk footprint = 25,000 * (512 + 512) bytes = 25,600,000 bytes ? 24.4 MiB
> - Double the estimated disk footprint to determine the recommended journal size

5.3.4.9. Resize the Journal

To avoid the fatal condition caused by a full message store, it is possible to resize the journal. This is achieved using a utility that can read the store and transfer all active records in it into a new larger journal. This can only be done when the store is not active (ie broker is not running). The resize utility is located in `/usr/libexec/qpid/` and to use it the Python path must include this directory.

The tools involved in resizing the journal, are `resize` and `store_chk`. The `resize` command resizes a store to make it bigger or smaller then transfers all outstanding records from the old store to the new store. If the records will not fit into the file, there will be an error message. The old store remains saved in a subdirectory. The `store_chk` command analyzes a store, and shows the outstanding records and transactions.

Report a bug

Chapter 6. Initial Tuning

6.1. Run the JMS Client with real-time Java

To achieve more deterministic behavior, the JMS Client can be run in a Realtime Java environment.

1. The client must be run on a realtime operating system, and supported by your realtime java vendor. Red Hat supports only Sun and IBM implementations.
2. Place the realtime .jar files provided by your vendor in the classpath.
3. Set the following JVM argument:

   ```
   -Dqpid.thread_factory="org.apache.qpid.thread.RealtimeThreadFactory"
   ```

   This ensures that the JMS Client will use `javax.realtime.RealtimeThreads` instead of `java.lang.Threads`.

   Optionally, the priority of the Threads can be set using:

   ```
   -Dqpid.rt_thread_priority=30
   ```

   By default, the priority is set at 20.
4. Based on your workload, the JVM will need to be tuned to achieve the best results. Refer to your vendor's JVM tuning guide for more information.

Report a bug

6.2. qpid-perftest

`qpid-perftest` is a command-line utility for measuring throughput. It is supplied as part of the `qpid-cpp-client-devel` package.

Running `qpid-perftest` provides statistics on the maximum performance of your Messaging Server. You can compare the results of `qpid-perftest` with the performance of your application to determine whether your application or the Messaging Server is a performance bottleneck.

`qpid-perftest --help` provides further information on running the utility.

Report a bug

6.3. qpid-latency-test

`qpid-latency-test` is a command-line utility for measuring latency. It is supplied as part of the `qpid-cpp-client-devel` package.

Running `qpid-latency-test` provides statistics on the performance of your Messaging Server. You can compare the results of `qpid-latency-test` with the performance of your application to determine whether your application or the Messaging Server is a performance bottleneck.

`qpid-latency-test --help` provides further information on running the utility.

Report a bug

6.4. Infiniband

6.4.1. Using Infiniband

MRG Messaging connections can use Infiniband, which provides high speed point-to-point bidirectional serial links that can be faster and have much lower latency than TCP connections.

Report a bug

6.4.2. Prerequisites for using Infiniband

The machines running the server and client must each have Infiniband properly installed. In particular:

- The kernel driver and the user space driver for your Infiniband hardware must both be installed.
- Allocate lockable memory for Infiniband.

  By default, the operating system can swap out all user memory. Infiniband requires lockable memory, which can not be swapped out. Each connection requires 8 Megabytes (8192 bytes) of lockable memory.

  To allocate lockable memory, edit `/etc/security/limits.conf` to set the limit, which is the maximum amount of lockable memory that a given process can allocate.
- The Infiniband interface must be configured to allow IP over Infiniband. This is used for RDMA connection management.

Report a bug

6.4.3. Configure Infiniband on the Messaging Server

Prerequisites

- The package `qpid-cpp-server-rdma` must be installed for Qpid to use RDMA.
- The RDMA plugin, `rdma.so`, must be present in the `plugins` directory.

Procedure 6.1. Configure Infiniband on the Messaging Server

- **Allocate lockable memory for Infiniband**

  Edit `/etc/security/limits.conf` to allocate lockable memory for Infiniband.

  For example, if the user running the server is qpidd, and you wish to support 64 connections (64*8192=524288), add these entries:

```
qpidd soft memlock 524288
qpidd hard memlock 524288
```

Report a bug

6.4.4. Configure Infiniband on a Messaging Client

Prerequisites

- The package `qpid-cpp-client-rdma` must be installed.

Procedure 6.2. Configure Infiniband on a Messaging Client

- **Allocate lockable memory for Infiniband**

  Edit `/etc/security/limits.conf` to allocate lockable memory.

To set a limit for all users, for example supporting 16 connections (16*8192=32768), add this entry:

```
* soft memlock 32768
```

If you want to set a limit for a particular user, use the UID for that user when setting the limits:

```
andrew soft memlock 32768
```

Report a bug

Chapter 7. Logging

7.1. Logging in C++

The Qpidd broker and C++ clients can both use environment variables to enable logging. Linux and Windows systems use the same named environment variables and values.

1. Use `QPID_LOG_ENABLE` to set the level of logging you are interested in (*`trace`*, *`debug`*, *`info`*, *`notice`*, *`warning`*, *`error`*, or *`critical`*):

```
export QPID_LOG_ENABLE="warning+"
```

2. The Qpidd broker and C++ clients use `QPID_LOG_OUTPUT` to determine where logging output should be sent. This is either a file name or the special values *`stderr`*, *`stdout`*, or *`syslog`*:

```
export QPID_LOG_TO_FILE="/tmp/myclient.out"
```

3. From a Windows command prompt, use the following command format to set the environment variables:

```
set QPID_LOG_ENABLE=warning+
set QPID_LOG_TO_FILE=D:\tmp\myclient.out
```

Report a bug

7.2. Change Broker Logging Verbosity

Changes

When running qpidd from command line, use `--log-enable` option with the syntax:

```
--log-enable LEVEL[+][:PATTERN]
```

When using a configuration file (`/etc/qpidd.conf` by default), use the line:

```
log-enable=LEVEL[+][:PATTERN]
```

Notes

- `LEVEL` is one of: `trace debug info notice warning error critical`.
- The "+" means log the given severity and any higher severity (without the plus, logging of the given severity only will be enabled).
- `PATTERN` is the scope of the logging change.
- The string in `PATTERN` is matched against the fully-qualified name of the C++ function with the logging statement.
- To see the fully-qualified name of the C++ function with the logging statement, either check the source code or add to the qpid configuration the `log-function=yes` option to force qpid broker to log such message.
- So e.g. `--log-enable debug+:cluster` matches everything in the `qpid::cluster` module, while e.g. `--log-enable debug+:broker::Queue::consumeNextMessage` will enable logging of one particular method only (the `consumeNextMessage` method in the given namespace in this example).

- PATTERN is often set to the module one needs to debug, like acl, amqp_0_10, broker, cluster, management or store.
- The option can be used multiple times.
- Be aware that having just one option like "`log-enable=debug+:cluster`" enables debug logs of cluster stuff, but does not produce any other logs; to add some more verbose logging, one should add such option like the above plus "artificially" add there the default value: `log-enable=info+`

Report a bug

7.3. Tracking Object Lifecycles

Changes

MRG 2.3 introduces a new log category, `[Model]`, to track the creation, destruction, and major state changes to Connection, Session, and Subscription objects, and to Exchange, Queue, and Binding objects.

From this set of log messages you can determine which user from which client system address created a connection, what sessions were created on that connection, and what subscriptions were created on those sessions.

Similarly, the exchange-binding-queue objects have enough in their log messages to correlate the interactions between them. The log message for the destruction of an object contains a record of all the management statistics kept for that object. Working through the log records you can attribute broker usage back to specific users.

At debug log level are log entries that mirror the corresponding management events. Debug level statements include user names, remote host information, and other references using the user-specified names for the referenced objects.

At `trace` log level are log entries that track the construction and destruction of managed resources. Trace level statements identify the objects using the internal management keys. The trace statement for each deleted object includes the management statistics for that object.

Enabling the Model log

- Use the switch: `--log-enable trace+:Model` to receive both flavors of log.
- Use the switch: `--log-enable debug+:Model` for a less verbose log.

Managed Objects in the logs

All managed objects are included in the trace log. The debug log has information for: `Connection, Queue, Exchange, Binding, Subscription`.

The following are actual log file data sorted and paired with the corresponding management Event captured with `qpid-printevents`.

1. Connection

Create connection

```
event: Fri Jul 13 17:46:23 2012 org.apache.qpid.broker:clientConnect
rhost=[::1]:5672-[::1]:34383 user=anonymous
debug: 2012-07-13 13:46:23 [Model] debug Create connection. user:anonymous
rhost:[::1]:5672-[::1]:34383
trace: 2012-07-13 13:46:23 [Model] trace Mgmt create connection. id:[::1]:5672-
[::1]:34383
```

Delete connection

```
event: Fri Jul 13 17:46:23 2012 org.apache.qpid.broker:clientDisconnect
rhost=[::1]:5672-[::1]:34383 user=anonymous
debug: 2012-07-13 13:46:23 [Model] debug Delete connection. user:anonymous
rhost:[::1]:5672-[::1]:34383
trace: 2012-07-13 13:46:29 [Model] trace Mgmt delete connection. id:[::1]:5672-
[::1]:34383
Statistics: {bytesFromClient:1451, bytesToClient:892, closing:False,
framesFromClient:25, framesToClient:21, msgsFromClient:1, msgsToClient:1}
```

2. Session

Create session

```
event: TBD
debug: TBD
trace: 2012-07-13 13:46:09 [Model] trace Mgmt create session. id:18f52c22-efc5-
4c2f-bd09-902d2a02b948:0
```

Delete session

```
event: TBD
debug: TBD
trace: 2012-07-13 13:47:13 [Model] trace Mgmt delete session. id:18f52c22-efc5-
4c2f-bd09-902d2a02b948:0
Statistics: {TxnCommits:0, TxnCount:0, TxnRejects:0, TxnStarts:0, clientCredit:0,
unackedMessages:0}
```

3. Exchange

Create exchange

```
event: Fri Jul 13 17:46:34 2012 org.apache.qpid.broker:exchangeDeclare
disp=created exName=myE exType=topic durable=False args={} autoDel=False
rhost=[::1]:5672-[::1]:34384 altEx= user=anonymous
debug: 2012-07-13 13:46:34 [Model] debug Create exchange. name:myE user:anonymous
rhost:[::1]:5672-[::1]:34384 type:topic alternateExchange: durable:F
trace: 2012-07-13 13:46:34 [Model] trace Mgmt create exchange. id:myE
```

Delete exchange

```
event: Fri Jul 13 18:19:33 2012 org.apache.qpid.broker:exchangeDelete exName=myE
rhost=[::1]:5672-[::1]:37199 user=anonymous
debug: 2012-07-13 14:19:33 [Model] debug Delete exchange. name:myE user:anonymous
rhost:[::1]:5672-[::1]:37199
trace: 2012-07-13 14:19:42 [Model] trace Mgmt delete exchange. id:myE
Statistics: {bindingCount:0, bindingCountHigh:0, bindingCountLow:0, byteDrops:0,
byteReceives:0, byteRoutes:0, msgDrops:0, msgReceives:0, msgRoutes:0,
producerCount:0, producerCountHigh:0, producerCountLow:0}
```

4. Queue

Create queue

```
event: Fri Jul 13 18:19:35 2012 org.apache.qpid.broker:queueDeclare disp=created
durable=False args={} qName=myQ autoDel=False rhost=[::1]:5672-[::1]:37200 altEx=
excl=False user=anonymous
debug: 2012-07-13 14:19:35 [Model] debug Create queue. name:myQ user:anonymous
rhost:[::1]:5672-[::1]:37200 durable:F owner:0 autodelete:F alternateExchange:
trace: 2012-07-13 14:19:35 [Model] trace Mgmt create queue. id:myQ
```

Delete queue

```
event: Fri Jul 13 18:19:37 2012 org.apache.qpid.broker:queueDelete user=anonymous
qName=myQ rhost=[::1]:5672-[::1]:37201
debug: 2012-07-13 14:19:37 [Model] debug Delete queue. name:myQ user:anonymous
rhost:[::1]:5672-[::1]:37201
trace: 2012-07-13 14:19:42 [Model] trace Mgmt delete queue. id:myQ
Statistics: {acquires:0, bindingCount:0, bindingCountHigh:0, bindingCountLow:0,
byteDepth:0, byteFtdDepth:0, byteFtdDequeues:0, byteFtdEnqueues:0,
bytePersistDequeues:0, bytePersistEnqueues:0, byteTotalDequeues:0,
byteTotalEnqueues:0, byteTxnDequeues:0, byteTxnEnqueues:0, consumerCount:0,
consumerCountHigh:0, consumerCountLow:0, discardsLvq:0, discardsOverflow:0,
discardsPurge:0, discardsRing:0, discardsSubscriber:0, discardsTtl:0,
flowStopped:False, flowStoppedCount:0, messageLatencyAvg:0, messageLatencyCount:0,
messageLatencyMax:0, messageLatencyMin:0, msgDepth:0, msgFtdDepth:0,
msgFtdDequeues:0, msgFtdEnqueues:0, msgPersistDequeues:0, msgPersistEnqueues:0,
msgTotalDequeues:0, msgTotalEnqueues:0, msgTxnDequeues:0, msgTxnEnqueues:0,
releases:0, reroutes:0, unackedMessages:0, unackedMessagesHigh:0,
unackedMessagesLow:0}
```

5. Binding

Create binding

```
event: Fri Jul 13 17:46:45 2012 org.apache.qpid.broker:bind exName=myE args={}
qName=myQ user=anonymous key=myKey rhost=[::1]:5672-[::1]:34385
debug: 2012-07-13 13:46:45 [Model] debug Create binding. exchange:myE queue:myQ
key:myKey user:anonymous rhost:[::1]:5672-[::1]:34385
trace: 2012-07-13 13:46:23 [Model] trace Mgmt create binding.
id:org.apache.qpid.broker:exchange:,org.apache.qpid.broker:queue:myQ,myQ
```

Delete binding

```
event: Fri Jul 13 17:47:06 2012 org.apache.qpid.broker:unbind user=anonymous
exName=myE qName=myQ key=myKey rhost=[::1]:5672-[::1]:34386
debug: 2012-07-13 13:47:06 [Model] debug Delete binding. exchange:myE queue:myQ
key:myKey user:anonymous rhost:[::1]:5672-[::1]:34386
trace: 2012-07-13 13:47:09 [Model] trace Mgmt delete binding.
id:org.apache.qpid.broker:exchange:myE,org.apache.qpid.broker:queue:myQ,myKey
Statistics: {msgMatched:0}
```

6. Subscription

Create subscription

```
event: Fri Jul 13 18:19:28 2012 org.apache.qpid.broker:subscribe dest=0 args={}
qName=b78b1818-7a20-4341-a253-76216b40ab4a:0.0 user=anonymous excl=False
rhost=[::1]:5672-[::1]:37198
debug: 2012-07-13 14:19:28 [Model] debug Create subscription. queue:b78b1818-7a20-
4341-a253-76216b40ab4a:0.0 destination:0 user:anonymous rhost:[::1]:5672-
[::1]:37198 exclusive:F
trace: 2012-07-13 14:19:28 [Model] trace Mgmt create subscription.
id:org.apache.qpid.broker:session:b78b1818-7a20-4341-a253-
76216b40ab4a:0,org.apache.qpid.broker:queue:b78b1818-7a20-4341-a253-
76216b40ab4a:0.0,0
```

Delete subscription

```
event: Fri Jul 13 18:19:28 2012 org.apache.qpid.broker:unsubscribe dest=0
rhost=[::1]:5672-[::1]:37198 user=anonymous
debug: 2012-07-13 14:19:28 [Model] debug Delete subscription. destination:0
user:anonymous rhost:[::1]:5672-[::1]:37198
trace: 2012-07-13 14:19:32 [Model] trace Mgmt delete subscription.
id:org.apache.qpid.broker:session:b78b1818-7a20-4341-a253-
76216b40ab4a:0,org.apache.qpid.broker:queue:b78b1818-7a20-4341-a253-
76216b40ab4a:0.0,0
Statistics: {delivered:1}
```

Report a bug

Chapter 8. Security

8.1. Simple Authentication and Security Layer - SASL

8.1.1. SASL defined

MRG Messaging uses Simple Authentication and Security Layer (SASL) for identifying and authorizing incoming connections to the broker, as mandated in the AMQP specification. SASL provides a variety of authentication methods. MRG Messaging clients (with the exception of the JMS client) and the broker use the Cyrus SASL library to allow for a full SASL implementation.

Report a bug

8.1.2. SASL Support in Windows Clients

The Windows Qpid C++ client supports only `ANONYMOUS` and `PLAIN` authentication mechanisms.

This is likely to change in a future release.

Report a bug

8.1.3. SASL Mechanisms

The SASL authentication mechanisms allowed by the broker are controlled by the file `/etc/sasl2/qpidd.conf` on the broker. To narrow the allowed mechanisms to a smaller subset, edit this file and remove mechanisms.

Report a bug

8.1.4. Configure SASL using a Local Password File

The MRG Messaging client libraries implement the default SASL PLAIN authentication mechanism by default. The allowed SASL mechanisms are controlled by the file `/etc/sasl2/qpidd.conf` on the broker. Edit this file to narrow the allowed SASL authentication mechanisms.

Important

The PLAIN authentication mechanism sends passwords in cleartext. If using this mechanism, for complete security using Security Services Library (SSL) is recommended.

Note

To use SSL in python Qpid clients using a version earlier than Python 2.6, you need to install the *python-ssl* package from the Extra Packages for Enterprise Linux (EPEL) repository.

Procedure 8.1. Configure SASL using a Local Password File

1. Add new users to the database by using the `saslpasswd2` command. The User ID for

authentication and ACL authorization uses the form *user-id@domain*.

Ensure that the correct realm has been set for the broker. This can be done by editing the configuration file or using the `-u` option. The default realm for the broker is *QPID*.

```
# saslpasswd2 -f /var/lib/qpidd/qpidd.sasldb -u QPID new_user_name
```

2. Existing user accounts can be listed by using the `-f` option:

```
# sasldblistusers2 -f /var/lib/qpidd/qpidd.sasldb
```

Note

The user database at `/var/lib/qpidd/qpidd.sasldb` is readable only by the `qpidd` user. If you start the broker from a user other than the `qpidd` user, you will need to either modify the configuration file, or turn authentication off.

3. To switch authentication on or off, use the `auth yes|no` option when you start the broker:

```
# /usr/sbin/qpidd --auth yes

# /usr/sbin/qpidd --auth no
```

You can also set authentication to be on or off by adding the appropriate line to to the `/etc/qpidd.conf` configuration file:

```
auth=no

auth=yes
```

The SASL configuration file is in `/etc/sasl2/qpidd.conf` for Red Hat Enterprise Linux.

Report a bug

8.1.5. Configure SASL with ACL

1. The ACL module is loaded by default. You can check that it is loaded by running the `qpidd --help` command and checking the output for ACL options:

```
$ qpidd --help
...[output truncated]...
ACL Options:
--acl-file FILE (policy.acl)  The policy file to load from, loaded from data
dir
--connection-limit-per-user N (0)  The maximum number of connections allowed
per user. 0 implies no limit.
--connection-limit-per-ip N (0)    The maximum number of connections allowed
per host IP address. 0 implies no limit.
```

2. To start using the ACL, specify the path and filename using the `--acl-file` command. The filename should have a `.acl` extension:

```
$ qpidd --acl-file ./aclfilename.acl
```

3. Optionally, you can limit the number of active connections per user with the `--connection-limit-per-user` and `--connection-limit-per-ip` commands. These limits can only be enforced if the `--acl-file` command is specified.
4. You can now view the file with the `cat` command and edit it in your preferred text editor.

If the path and filename is not found, qpidd will fail to start.

Report a bug

8.1.6. Configure Kerberos 5

Kerberos uses the GSSAPI (Generic Security Services Application Program Interface) authentication mechanism on the broker to authenticate with a Kerberos server.

Both the MRG Messaging broker and users are principals of the Kerberos server, which means that they are both clients of the Kerberos authentication services.

Note

The following instructions make use of example domain names and Kerberos realms. To follow these instructions you must have a Kerberos server configured and use the appropriate domain names and Kerberos realm for your network environment.

To use Kerberos, both the broker and each user must be authenticated on the Kerberos server:

1. Install the Kerberos workstation software and Cyrus SASL GSSAPI on each machine that runs a qpidd broker or a qpidd messaging client:

   ```
   $ sudo yum install cyrus-sasl-gssapi krb5-workstation
   ```

2. Change the mech_list line in /etc/sasl2/qpidd.conf to:

   ```
   mech_list: GSSAPI
   ```

3. Add the following lines to /etc/qpidd.conf:

   ```
   auth=yes
   realm=QPID
   ```

4. Make sure that the Qpid broker is registered in the Kerberos database.

   Traditionally, a Kerberos principal is divided into three parts: the primary, the instance, and the realm. A typical Kerberos V5 has the format primary/instance@REALM. For a broker, the primary is qpidd, the instance is the fully qualified domain name, and the REALM is the Kerberos domain realm. By default, this realm is QPID, but a different realm can be specified in qpid.conf per the following example.

   ```
   realm=EXAMPLE.COM
   ```

   For instance, if the fully qualified domain name is dublduck.example.com and the Kerberos domain realm is EXAMPLE.COM, then the principal name is qpidd/dublduck.example.com@EXAMPLE.COM.

   ```
   FDQN=`hostname --fqdn`
   REALM="EXAMPLE.COM"
   kadmin -r $REALM  -q "addprinc -randkey -clearpolicy qpidd/$FQDN"
   ```

   Now create a Kerberos keytab file for the broker. The broker must have read access to the keytab file. The following script creates a keytab file and allows the broker read access:

```
QPIDD_GROUP="qpidd"
kadmin -r $REALM  -q "ktadd -k /etc/qpidd.keytab qpidd/$FQDN@$REALM"
chmod g+r /etc/qpidd.keytab
chgrp $QPIDD_GROUP /etc/qpidd.keytab
```

The default location for the keytab file is `/etc/krb5.keytab`. If a different keytab file is used, the KRB5_KTNAME environment variable must contain the name of the file as the following example shows.

```
export KRB5_KTNAME=/etc/qpidd.keytab
```

If this is correctly configured, you can now enable Kerberos support on the broker by setting the `auth` and `realm` options in `/etc/qpidd.conf`:

```
CDATA[# /etc/qpidd.conf
auth=yes
realm=EXAMPLE.COM
```

Restart the broker to activate these settings.

5. Make sure that each Qpid user is registered in the Kerberos database, and that Kerberos is correctly configured on the client machine. The Qpid user is the account from which a Qpid messaging client is run. If it is correctly configured, the following command should succeed:

```
$ kinit user@REALM.COM
```

6. **Additional configuration for Java JMS clients**

   Java JMS clients require a few additional steps.

   a. The Java JVM must be run with the following arguments:

      -Djavax.security.auth.useSubjectCredsOnly=false
      Forces the SASL GASSPI client to obtain the Kerberos credentials explicitly instead of obtaining from the "subject" that owns the current thread.

      -Djava.security.auth.login.config=myjas.conf
      Specifies the jass configuration file. Here is a sample JASS configuration file:

      ```
      com.sun.security.jgss.initiate {
          com.sun.security.auth.module.Krb5LoginModule required
      useTicketCache=true;
      };
      ```

      -Dsun.security.krb5.debug=true
      Enables detailed debug info for troubleshooting

   b. The client Connection URL must specify the following Kerberos-specific broker properties:

      - `sasl_mechs` must be set to `GSSAPI`.
      - `sasl_protocol` must be set to the principal for the qpidd broker, e.g. `qpidd/`
      - `sasl_server` must be set to the host for the SASL server, e.g. `sasl.com`.

      Here is a sample connection URL for a Kerberos connection:

      ```
      amqp://guest@clientid/testpath?brokerlist='tcp://localhost:5672?
      sasl_mechs='GSSAPI'&sasl_protocol='qpidd'&sasl_server='<server-host-
      name>''
      ```

Report a bug

8.2. Configuring TLS/SSL

8.2.1. Encryption Using SSL

Encryption and certificate management for `qpidd` is provided by Mozilla's Network Security Services Library (NSS).

See Also:

- Appendix B, *OpenSSL Certificate Reference*

Report a bug

8.2.2. Enable SSL on the Broker

1. You will need a certificate that has been signed by a Certification Authority (CA). This certificate will also need to be trusted by your client. If you require client authentication in addition to server authentication, the clients certificate will also need to be signed by a CA and trusted by the broker.

   In the broker, SSL is provided through the `ssl.so` module. This module is installed and loaded by default in MRG Messaging. To enable the module, you need to specify the location of the database containing the certificate and key to use. This is done using the `ssl-cert-db` option.

   The certificate database is created and managed by the Mozilla Network Security Services (NSS) `certutil` tool. Information on this utility can be found on the Mozilla website, including tutorials on setting up and testing SSL connections. The certificate database will generally be password protected. The safest way to specify the password is to place it in a protected file, use the password file when creating the database, and specify the password file with the `ssl-cert-password-file` option when starting the broker.

   The following script shows how to create a certificate database using certutil:

   ```
   mkdir ${CERT_DIR}
   certutil -N -d ${CERT_DIR} -f ${CERT_PW_FILE}
   certutil -S -d ${CERT_DIR} -n ${NICKNAME} -s "CN=${NICKNAME}"" -t "CT,," -x
   -f ${CERT_PW_FILE} -z /usr/bin/certutil
   ```

   When starting the broker, set `ssl-cert-password-file` to the value of `${CERT_PW_FILE}`, set `ssl-cert-db` to the value of `${CERT_DIR}`, and set `ssl-cert-name` to the value of `${NICKNAME}`.

2. The following SSL options can be used when starting the broker:

   `--ssl-use-export-policy`
   Use NSS export policy

   `--ssl-cert-password-file` *`PATH`*
   Required. Plain-text file containing password to use for accessing certificate database.

   `--ssl-cert-db` *`PATH`*
   Required. Path to directory containing certificate database.

   `--ssl-cert-name` *`NAME`*
   Name of the certificate to use. Default is `localhost.localdomain`.

   `--ssl-port` *`NUMBER`*
   Port on which to listen for SSL connections. If no port is specified, port 5671 is used.

If the SSL port chosen is the same as the port for non-SSL connections (i.e. if the `--ssl-port` and `--port` options are the same), both SSL encrypted and unencrypted connections can be established to the same port. However in this configuration there is no support for IPv6.

`--ssl-require-client-authentication`
Require SSL client authentication (i.e. verification of a client certificate) during the SSL handshake. This occurs before SASL authentication, and is independent of SASL.

This option enables the `EXTERNAL` SASL mechanism for SSL connections. If the client chooses the `EXTERNAL` mechanism, the client's identity is taken from the validated SSL certificate, using the `CN`, and appending any `DC`'s to create the domain. For instance, if the certificate contains the properties `CN=bob`, `DC=acme`, `DC=com`, the client's identity is `bob@acme.com`.

If the client chooses a different SASL mechanism, the identity take from the client certificate will be replaced by that negotiated during the SASL handshake.

`--ssl-sasl-no-dict`
Do not accept SASL mechanisms that can be compromised by dictionary attacks. This prevents a weaker mechanism being selected instead of `EXTERNAL`, which is not vulnerable to dictionary attacks.

Also relevant is the `--require-encryption` broker option. This will cause `qpidd` to only accept encrypted connections.

Report a bug

8.2.3. Export an SSL Certificate for Clients

When SSL is enabled on a server, the clients require a copy of the SSL certificate to establish a secure connection.

The following example commands can be used to export a client certificate and the private key from the broker's NSS database:

```
pk12util -o <p12exportfile> -n <certname> -d <certdir> -w <p12filepwfile>

openssl pkcs12 -in <p12exportfile> -out <clcertname> -nodes -clcerts -passin
pass:<p12pw>
```

For more information on SSL commands and options, refer to the OpenSSL Documentation. On Red Hat Enterprise Linux type: `man openssl`.

Report a bug

8.2.4. Enable SSL in C++ Clients

In C++ clients, SSL is implemented in the `sslconnector.so` module. This module is installed and loaded by default in MRG Messaging.

The following options can be specified for C++ clients using environment variables:

Table 8.1. SSL Client Environment Variables for C++ clients

SSL Client Options for C++ clients	
`QPID_SSL_USE_EXPORT_POLICY`	Use NSS export policy
`QPID_SSL_CERT_PASSWORD_FILE` *PATH*	File containing password to use for accessing certificate database
`QPID_SSL_CERT_DB` *PATH*	Path to directory containing certificate database
`QPID_SSL_CERT_NAME` *NAME*	Name of the certificate to use. When SSL client authentication is enabled, a certificate name should normally be provided.

When using SSL connections, clients must specify the location of the certificate database, a directory that contains the client's certificate and the public key of the Certificate Authority. This can be done by setting the environment variable `QPID_SSL_CERT_DB` to the full pathname of the directory. If a connection uses SSL client authentication, the client's password is also needed - the password should be placed in a protected file, and the `QPID_SSL_CERT_PASSWORD_FILE` variable should be set to the location of the file containing this password.

To open an SSL enabled connection in the Qpid Messaging API, set the *transport* connection option to *ssl*.

See Also:

- Section 8.2.3, “Export an SSL Certificate for Clients”
- Appendix B, *OpenSSL Certificate Reference*

Report a bug

8.2.5. Enable SSL in Java Clients

1. For both server and client authentication, import the trusted CA to your trust store and keystore and generate keys for them. Create a certificate request using the generated keys and then create a certificate using the request. You can then import the signed certificate into your keystore. Pass the following arguments to the Java JVM when starting your client:

```
-Djavax.net.ssl.keyStore=/home/bob/ssl_test/keystore.jks
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=/home/bob/ssl_test/certstore.jks
-Djavax.net.ssl.trustStorePassword=password
```

2. For server side authentication only, import the trusted CA to your trust store and pass the following arguments to the Java JVM when starting your client:

```
-Djavax.net.ssl.trustStore=/home/bob/ssl_test/certstore.jks
-Djavax.net.ssl.trustStorePassword=password
```

3. Java clients must use the SSL option in the connection URL to enable SSL encryption, per the following example.

```
amqp://username:password@clientid/test?brokerlist='tcp://localhost:5672?
ssl='true''
```

4. If you need to debug problems in an SSL connection, enable Java's SSL debugging by passing the argument `-Djavax.net.debug=ssl` to the Java JVM when starting your client.

See Also:

- Section 8.2.3, “Export an SSL Certificate for Clients”
- Appendix B, *OpenSSL Certificate Reference*

Report a bug

8.2.6. Enable SSL in Python Clients

Changes

For proper SSL functionality on Red Hat Enterprise Linux 5, it is recommended to install the `python-ssl` package from the Extra Packages for Enterprise Linux (EPEL) repository.

To use SSL with the Python client *either*:

1. Use a URL of the form `amqps://<host>:<port>`, where *host* is the brokers hostname and *port* is the SSL port (usually 5671), *or*
2. Set the '`transport`' attribute of the connection to "`ssl`".

The Python client has some limitations in SSL functionality. In MRG versions up to 2.2 it does not support client authentication with SSL.

MRG 2.3
MRG version 2.3 supports client authentication with SSL, with some caveats. Server authentication is not supported.
The caveats for SSL support in the MRG 2.3 Python client are:

- The server's SSL certificate is not validated against the chain of trust in the client's certificate store.
- The server's SSL certificate is not matched against the connection hostname.
- When using EXTERNAL sasl mechanism for authentication, the client's identity is not read from the client's certificate. You must provide the client name in the connection.

The QPID Python client accepts the following SSL-related configuration parameters:

- `ssl_certfile` - the path to a file that contains the PEM-formatted certificate used to identify the local side of the connection (the client). This is needed if the server requires client-side authentication.
- `ssl_keyfile` - In some cases the client's private key is stored in the same file as the certificate (i.e. ssl_certfile). If the `ssl_certfile` does not contain the client's private key, this parameter must be set to the path to a file containing the private key in PEM file format.
- These parameters are passed as arguments to the `qpid.Connection()` object when it is constructed. For example:

```
Connection("amqps://client@127.0.0.1:5671", ssl_certfile="/path/to/certfile",
ssl_keyfile="/path/to/keyfile")
```

See Also:

- Section 8.2.3, “Export an SSL Certificate for Clients”
- Appendix B, *OpenSSL Certificate Reference*

Report a bug

8.3. Authorization

8.3.1. Access Control List (ACL)

In MRG Messaging, Authorization specifies which actions can be performed by each authenticated user using an Access Control List (ACL).

Report a bug

8.3.2. Default ACL File

In versions up to 2.2, the location of the default ACL file is `/etc/qpidd.acl`.

From version 2.3, the default ACL file is relocated to `/etc/qpidd/qpidd.acl`. Unmodified existing installations will continue to use the previous ACL file and location, while any new installations will use the new default location and file.

Report a bug

8.3.3. Load an Access Control List (ACL)

Use the `--acl-file` command to load the access control list. The filename should have a `.acl` extension:

```
$ qpidd --acl-file ./aclfilename.acl
```

Report a bug

8.3.4. Reloading the ACL

You can reload the ACL without restarting the broker using a QMF method, either using `qpid-tool` or from program code.

Reload the ACL using `qpid-tool`

You need to use `qpid-tool` with a account with sufficient privileges to reload the ACL.

1. Start `qpid-tool`:

```
$ qpid-tool admin/mysecretpassword@mybroker:5672
Management Tool for QPID
qpid:
```

2. Check the ACL list to obtain the object ID:

```
qpid: list acl
Object Summary:
    ID   Created   Destroyed   Index
    ================================
    103  12:57:41  -           116
```

3. Optionally, you can examine the ACL:

```
qpid: show 103
Object of type: org.apache.qpid.acl:acl:_data(23510fc1-dc51-a952-39c2-
e18475c1677e)
    Attribute               103
    ================================================
    brokerRef               116
    policyFile              /tmp/reload.acl
    enforcingAcl            True
    transferAcl             False
    lastAclLoad             Tue Oct 30 12:57:41 2012
    maxConnectionsPerIp     0
    maxConnectionsPerUser   0
    maxQueuesPerUser        0
    aclDenyCount            0
    connectionDenyCount     0
    queueQuotaDenyCount     0
```

4. To reload the ACL, call the reload method of the ACL object:

```
qpid: call 103 reloadACLFile
qpid: OK (0) - {}
```

Reload ACL from program code

The broker ACL can be reloaded at runtime by calling a QMF method.

The following code calls the appropriate QMF method to reload the ACL:

Python

```
import qmf.console
qmf = qmf.console.Session()
qmf_broker = qmf.addBroker('localhost:5672')
acl = qmf.getObjects(_class="acl")[0]
result = acl.reloadACLFile()
print result
```

Note that the server must be started with ACL enabled for the reload operation to succeed.

Report a bug

8.3.5. Writing an Access Control List

1. Each line in an ACL file grants or denies specific rights to a user.
   a. If the last line in an ACL file is `acl deny all all`, the ACL uses *deny mode*, and only those rights that are explicitly allowed are granted:

      ```
      acl allow user@QPID all all
      acl deny all all
      ```

      On this server, deny mode is the default. `user@QPID` can perform any action, but nobody else can. In deny mode, denying rights to an action is redundant and has no effect.
   b. If the last line in an ACL file is `acl allow all all`, the ACL uses *allow mode*, and all rights are granted except those that are explicitly denied.

      ```
      acl deny user@QPID all all
      acl allow all all
      ```

On this server, allow mode is the default. The ACL allows everyone else to perform any action, but denies `user@QPID` all permissions. In allow mode, allowing rights to an action is redundant and has no effect.

2. ACL processing ends when one of the following lines is encountered:

```
acl allow all all
```

```
acl deny all all
```

Any lines after one of these statements will be ignored:

```
acl allow all all
acl deny user@QPID all all # This line is ignored !!!
```

3. ACL syntax allows fine-grained access rights for specific actions:

```
acl allow carlt@QPID create exchange name=carl.*
acl allow fred@QPID create all
acl allow all consume queue
acl allow all bind exchange
acl deny all all
```

4. An ACL file can define user groups, and assign permissions to them:

```
group admin ted@QPID martin@QPID
acl allow admin create all
acl deny all all
```

Report a bug

8.3.6. ACL Syntax

ACL rules must be on a single line and follow this syntax:

```
acl permission {<group-name>|<user-name>|"all"} {action|"all"} [object|"all"]
[property=<property-value>]
```

In ACL files, the following syntactic conventions apply:

- A line starting with the # character is considered a comment and is ignored.
- Empty lines and lines that contain only whitespace are ignored
- All tokens are case sensitive. *`name1`* is not the same as *`Name1`* and *`create`* is not the same as *`CREATE`*
- Group lists can be extended to the following line by terminating the line with the \ character
- Additional whitespace - that is, where there is more than one whitespace character - between and after tokens is ignored. Group and ACL definitions must start with either `group` or `acl` and with no preceding whitespace.
- All ACL rules are limited to a single line
- Rules are interpreted from the top of the file down until the name match is obtained; at which point processing stops.
- The keyword *`all`* matches all individuals, groups and actions
- The last line of the file - whether present or not - will be assumed to be `acl deny all all`. If present in the file, all lines below it are ignored.
- Names and group names may contain only *`a-z`*, *`A-Z`*, *`0-9`*, - and _
- Rules must be preceded by any group definitions they can use. Any name not defined as a group will be assumed to be that of an individual.

- Qpid fails to start if ACL file is not valid
- ACL rules can be reloaded at runtime by calling a QMF method

See Also:

- Section 8.3.4, “Reloading the ACL”

Report a bug

8.3.7. ACL Definition Reference

The following tables show the possible values for `permission`, `action`, `object`, and `property` in an ACL rules file.

Table 8.2. ACL Rules: permission

`allow`	Allow the action
`allow-log`	Allow the action and log the action in the event log
`deny`	Deny the action
`deny-log`	Deny the action and log the action in the event log

Table 8.3. ACL Rules: action

`consume`	Applied when subscriptions are created
`publish`	Applied on a per message basis on publish message transfers, this rule consumes the most resources
`create`	Applied when an object is created, such as bindings, queues, exchanges, links
`access`	Applied when an object is read or accessed
`bind`	Applied when objects are bound together
`unbind`	Applied when objects are unbound
`delete`	Applied when objects are deleted
`purge`	Similar to delete but the action is performed on more than one object
`update`	Applied when an object is updated

Table 8.4. ACL Rules: object

`queue`	A queue
`exchange`	An exchange
`broker`	The broker
`link`	A federation or inter-broker link
`method`	Management or agent or broker method

Table 8.5. ACL Rules: property

`name`	String. Object name, such as a queue name or exchange name.
`durable`	Boolean. Indicates the object is durable
`routingkey`	String. Specifies routing key
`passive`	Boolean. Indicates the presence of a *passive* flag
`autodelete`	Boolean. Indicates whether or not the object gets deleted when the connection is closed
`exclusive`	Boolean. Indicates the presence of an *exclusive* flag
`type`	String. Type of object, such as topic, fanout, or xml
`alternate`	String. Name of the alternate exchange
`queuename`	String. Name of the queue (used only when the object is something other than *queue*
`schemapackage`	String. QMF schema package name
`schemaclass`	String. QMF schema class name
`policytype`	String. The limit policy for a queue. Only used in rules for queue creation.
`maxqueuesize`	Integer. The largest value of the maximum queue size (in bytes) with which a queue is allowed to be created. Only used in rules for queue creation.
`maxqueuecount`	Integer. The largest value of the maximum queue depth (in messages) that a queue is allowed to be created. Only used in rules for queue creation.

Report a bug

8.3.8. Enforcing Queue Size Limits via ACL

In MRG 2.3 and above, the maximum queue size can be enforced via an ACL. This allows the administrator to disallow users from creating queues that could consume too many system resources.

CREATE QUEUE rules have ACL rules that limit the upper and lower bounds of both in-memory queue and on-disk queue store sizes.

Table 8.6. Queue Size ACL Rules

User Option	ACL Limit Property	Units
`qpid.max_size`	queuemaxsizelowerlimit	bytes
	queuemaxsizeupperlimit	bytes
`qpid.max_count`	queuemaxcountlowerlimit	messages
	queuemaxcountupperlimit	messages
`qpid.file_size`	filemaxsizelowerlimit	pages (64Kb per page)
	filemaxsizeupperlimit	pages (64Kb per page)
`qpid.file_count`	filemaxcountlowerlimit	files
	filemaxcountupperlimit	files

ACL Limit Properties are evaluated when the user presents one of the options in a CREATE QUEUE request. If the user's option is not within the limit properties for an ACL Rule that would allow the request, then the rule is matched with a Deny result.

Limit properties are ignored for Deny rules.

Example:

```
# Example of ACL specifying queue size constraints
# Note: for legibility this acl line has been split into multiple lines.
acl allow bob@QPID create queue name=q6 queuemaxsizelowerlimit=500000
                                        queuemaxsizeupperlimit=1000000
                                        queuemaxcountlowerlimit=200
                                        queuemaxcountupperlimit=300
```

These limits come into play when a queue is created as illustrated here:

```
int main(int argc, char** argv) {
const char* url = argc>1 ? argv[1] : "amqp:tcp:127.0.0.1:5672";
const char* address = argc>2 ? argv[2] :
    "message_queue; “
    “ { create: always, “
    “   node: “
    “   { type: queue, “
    “     x-declare: ”
    “     { arguments: “
    “       { qpid.max_count:101,”
    “         qpid.max_size:1000000”
    “       }”
    “     }”
    “   }”
    “ }";
std::string connectionOptions = argc > 3 ? argv[3] : "";

Connection connection(url, connectionOptions);
try {
    connection.open();
    Session session = connection.createSession();
    Sender sender = session.createSender(address);
...
```

This queue can also be created with the `qpid-config` command:

```
qpid-config add queue --max-queue-size=1000000 --max-queue-count=101
```

When the ACL rule is processed assume that the actor, action, object, and object name all match and so this allow rule matches for the allow or deny decision. However, the ACL rule is further constrained to limit 500000 <= max_size <= 1000000 and 200 <= max_count <= 300. Since the `queue_option max_count` is 101 then the size limit is violated (it is too low) and the allow rule is returned with a deny decision.

Note that it is not mandatory to set *both* an upper limit *and* a lower limit. It is possible to set only a lower limit, or only an upper limit.

Report a bug

8.3.9. Resource Quota Options

Changes

The maximum number of connections can be restricted:

Table 8.7. Resource Quota Options

Option	Description	Default Value
`--max-connections N`	total concurrent connections to the broker	500

Notes

- `--max-connections` is a qpid core limit and is enforced whether ACL is enabled or not.
- `--max-connections` is enforced per Broker. In a cluster of *N* nodes where all Brokers set the maximum connections to 20 the total number of allowed connections for the cluster will be *N**20.

ACL-based Quotas

To enable ACL-based quotas, an ACL file must be loaded:

Table 8.8. ACL Command-line Option

Option	Description	Default Value
`--acl-file FILE (policy.acl)`	The policy file to load from, loaded from data dir	

When an ACL file is loaded, the following ACL options can be specified at the command-line to enforce resource quotas:

Table 8.9. ACL-based Resource Quota Options

Option	Description	Default Value
`--connection-limit-per-user N`	`MRG 2.2+` The maximum number of connections allowed per user. 0 implies no limit.	0
`--connection-limit-per-ip N`	`MRG 2.2+` The maximum number of connections allowed per host IP address. 0 implies no limit.	0
`--max-queues-per-user N`	`MRG 2.3+` Total concurrent queues created by individual user	0

Notes

- In a cluster system the actual number of connections may exceed the connection quota value `N` by one less than the number of member nodes in the cluster. For example: in a 5-node cluster, with a limit of 20 connections, the actual number of connections can reach 24 before limiting takes place.
- Cluster connections are checked against the connection limit when they are established. The cluster connection is denied if a free connection is not available. After establishment, however, a cluster connection does not consume a connection.
- Allowed values for `N` are 0..65535.
- These limits are enforced per *cluster*.
- A value of zero (0) disables that option's limit checking.
- Per-user connections are identified by the authenticated user name.
- Per-ip connections are identified by the `<broker-ip><broker-port>-<client-`

ip><client-port> tuple which is also the management connection index.

- With this scheme host systems may be identified by several names such as localhost IPv4, 127.0.0.1 IPv4, or ::1 IPv6, and a separate set of connections is allowed for each name.
- Per-IP connections are counted regardless of the user credentials provided with the connections. An individual user may be allowed 20 connections but if the client host has a 5 connection limit then that user may connect from that system only 5 times.

Report a bug

8.3.10. Routing Key Wildcards

MRG 2.3 and above use the *Topic Exchange match logic* for ACL rules containing a routingkey property. These rules include:

- bind exchange <name> routingkey=*X*
- unbind exchange <name> routingkey=*X*
- publish exchange <name> routingkey=*X*

The routingkey property is now matched using the same logic as the Topic Exchange match. This allows administrators to express user limits in flexible terms that map to the namespace where routingkey values are used.

Wildcard matching and Topic Exchanges

In the binding key, # matches any number of period-separated terms, and * matches a single term.

So a binding key of #.news will match messages with subjects such as usa.news and germany.europe.news, while a binding key of *.news will match messages with the subject usa.news, but not germany.europe.news.

Example:

The following ACL rules:

```
acl allow-log uHash1@COMPANY publish exchange name=X routingkey=a.#.b
acl deny all all
```

Produce the following results when user uHash1@COMPANY publishes to exchange X:

Table 8.10.

routingkey in publish to exchange X	result
a.b	allow-log
a.x.b	allow-log
a..x.y.zz.b	allow-log
a.b.	deny
q.x.b	deny

Report a bug

8.3.11. User Name and Domain Name Symbol Substitution

MRG 2.3 and above have the ability to use a simple set of user name and domain name

substitution variables. This provides administrators with an easy way to define private or shared resources.

Symbol substitution is allowed in the Acl file anywhere that text is supplied for a property value.

In the following table an authenticated user `bob.user@QPID.COM` has his substitution keywords expanded.

Table 8.11.

Keyword	Expansion
${userdomain}	bob_user_QPID_COM
${user}	bob_user
${domain}	QPID_COM

The original name has the period "." and at symbol "@" characters translated into underscore "_". This allows substitutions to work when the substitution keyword is used in a routingkey in the ACL file.

Using Symbol Substitution and Wildcards in Routing Keys

The * symbol can be used a wildcard match for any number of characters in a single field in a routing key. For example:

```
acl allow user_group publish exchange name=users routingkey=${user}-delivery-*
```

The '#' symbol , when used in a routing key specification substitutes for any number of dotted subject name fields. User and Domain symbol substitutions can also be combined with the # wildcard symbol in routing keys, for example:

```
acl allow user_group bind exchange name=${user}-work2 routingkey=news.#.${user}
```

ACL Matching of Wildcards in Routing Keys

The ACL processing matches `${userdomain}` before matching either `${user}` or `${domain}`. In most circumstances ACL processing treats `${user}_${domain}` and `${userdomain}` as equivalent and the two forms may be used interchangeably. The exception to this is rules that specify wildcards within routing keys. In this case the combination `${user}_${domain}` will never match, and the form `${userdomain}` should be used.

For example, the following rule will never match:

```
acl allow all publish exchange name=X routingkey=${user}_${domain}.c
```

In that example, the rule will never match, as the ACL processor looks for routingkey `${userdomain}.c`.

ACL Symbol Substitution Example

Administrators can set up ACL rule files that allow every user to create a private exchange, a private queue, and a private binding between them. In this example the users are also allowed to create private backup exchanges, queues and bindings. This effectively provides limits to user's exchange, queue, and binding creation and guarantees that each user gets exclusive access to these resources.

```
#
# Create primary queue and exchange:
acl allow all create  queue    name=${user}-work alternate=${user}-work2
acl deny  all create  queue    name=${user}-work alternate=*
acl allow all create  queue    name=${user}-work
acl allow all create  exchange name=${user}-work alternate=${user}-work2
acl deny  all create  exchange name=${user}-work alternate=*
acl allow all create  exchange name=${user}-work
#
# Create backup queue and exchange
#
acl deny  all create  queue    name=${user}-work2 alternate=*
acl allow all create  queue    name=${user}-work2
acl deny  all create  exchange name=${user}-work2 alternate=*
acl allow all create  exchange name=${user}-work2
#
# Bind/unbind primary exchange
#
acl allow all bind   exchange name=${user}-work routingkey=${user}
queuename=${user}-work
acl allow all unbind exchange name=${user}-work routingkey=${user}
queuename=${user}-work
#
# Bind/unbind backup exchange
#
acl allow all bind   exchange name=${user}-work2 routingkey=${user}
queuename=${user}-work2
acl allow all unbind exchange name=${user}-work2 routingkey=${user}
queuename=${user}-work2
#

# deny mode
#
acl deny all all
```

Report a bug

8.3.12. ACL Definition Examples

Most ACL files begin by defining groups:

```
group admin ted@QPID martin@QPID
group user-consume martin@QPID ted@QPID
group group2 kim@QPID user-consume rob@QPID
group publisher group2 \
tom@QPID andrew@QPID debbie@QPID
```

Rules in an ACL file grant or deny specific permissions to users or groups:

```
acl allow carlt@QPID create exchange name=carl.*
acl allow rob@QPID create queue
acl allow guest@QPID bind exchange name=amq.topic routingkey=stocks.rht.#
acl allow user-consume create queue name=tmp.*

acl allow publisher publish all durable=false
acl allow publisher create queue name=RequestQueue
acl allow consumer consume queue durable=true
acl allow fred@QPID create all
acl allow bob@QPID all queue
acl allow admin all
acl allow all consume queue
acl allow all bind exchange
acl deny all all
```

In the previous example, the last line, `acl deny all all`, denies all authorizations that have not been specifically granted. This is the default, but it is useful to include it explicitly on the last line for the sake of clarity. If you want to grant all rights by default, you can specify `acl allow all all` in the last line.

Do not allow *guest* to access and log QMF management methods that could cause security breaches:

```
group allUsers guest@QPID
....
acl deny-log allUsers create link
acl deny-log allUsers access method name=connect
acl deny-log allUsers access method name=echo
acl allow all all
```

Report a bug

Chapter 9. High Availability

9.1. Clustering

9.1.1. Messaging Clusters

A *Messaging Cluster* is a group of brokers that act as a single broker. Every broker in a cluster has the same queues, exchanges, messages, and bindings. Messaging Clusters allow a client to *fail over* to a new broker and continue without any loss of messages if the current broker fails or becomes unavailable. Changes on any broker are replicated to all other brokers in the same Messaging Cluster, so if one broker fails, its clients can fail over to another broker without loss of state.

The brokers in a Messaging Cluster can run on the same host or on different hosts. Any number of messaging brokers can be run as one *cluster*, and brokers can be added to or removed from a cluster while it is in use. Two messaging brokers are in the same cluster if:

- They use the same OpenAIS `mcastaddr`, `mcastport`, and `bindnetaddr`, and
- They use the same cluster name.

High Availability Messaging Clusters are implemented using the OpenAIS Cluster Framework, which provides a reliable multicast protocol, tools, and infrastructure for implementing replicated services.

Note

Note that the `openais` package has been renamed to `corosync` in Red Hat Enterprise Linux 6.

Report a bug

9.1.2. Components of a Messaging Cluster

9.1.2.1. Red Hat Clustering Services

High Availability Messaging Clusters should be used together with the Red Hat High Cluster Suite (RHCS), also called Red Hat High Availability Add-On. Red Hat Cluster Suite provides a cluster manager (CMAN) which manages cluster quorum and cluster membership. Without CMAN, a network partition can cause lost or duplicated messages and unpredictable shutdown of Messaging brokers in the cluster.

CMAN keeps track of cluster quorum by monitoring the count of cluster nodes. If more than half the total cluster nodes are active, the cluster has quorum and can provide service. If a node loses contact with the rest of the cluster, the cluster loses quorum, and the broker on that node shuts down to avoid inconsistency. Clients then reconnect to a quorate broker. This ensures that only one group continues processing in the event of a network partition.

Cluster quorum prevents cluster "split-brain" conditions, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. A split-brain condition would allow each cluster instance to access cluster resources without knowledge of the other cluster instance, resulting in corrupted cluster integrity. For ideal operation, a cluster should

have an odd number of nodes.

Report a bug

9.1.3. Failover behavior in clients

9.1.3.1. Failover Behavior in Java JMS Clients

If a client is connected to a broker, the connection fails if the broker crashes or is killed. When a client's connection to a broker fails: (a) any sent messages that have been acknowledged by the sender are replicated to all brokers in the cluster; (b) any received messages that have not yet been acknowledged by the receiving client are requeued to all brokers, (c) the client API notifies the application of the failure by throwing an exception.

A client can be configured to automatically reconnect to another broker when it receives such an exception. Any messages that have been sent by the client, but not yet acknowledged as delivered, are resent. Any messages that have been read by the client, but not acknowledged, are delivered to the client.

In Java JMS clients, client failover is handled automatically if it is enabled in the connection. Any messages that have been sent by the client, but not yet acknowledged as delivered, are resent. Any messages that have been read by the client, but not acknowledged, are sent to the client.

You can configure a connection to use failover using the `failover` property:

```
connectionfactory.qpidConnectionfactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'&failover='failover_exchange'
```

`brokerlist` can take a semi-colon-separated list of brokers, like so:

```
brokerlist='<transport>://<host>[:<port>](?<param>=<value>)?
(&<param>=<value>)*(;<transport>://<host>[:<port>])?(?<param>=<value>)?
(&<param>=<value>)*'
```

For example:

```
brokerlist='tcp://ip1:5672;tcp://ip2:5672;tcp://ip3:5672?
ssl='true'&ssl_cert_alias='cert1'
```

Note that the broker option parameters are *per-broker*. Each broker in the list can have its own list of parameters, like so:

```
amqp://guest:guest@/test?failover='roundrobin?
cyclecount='2''&brokerlist='tcp://ip1:5672?
retries='5'&connectdelay='2000';tcp://ip2:5672?retries='5'&connectdelay='2000''
```

The failover property can take three values:

Failover Modes

failover_exchange
If the connection fails, fail over to any other broker in the cluster.

roundrobin
If the connection fails, remove head of `brokerlist` then fail over to the new broker now specified at head of list, until `brokerlist` is empty.

singlebroker

Failover is not supported; the connection is to a single broker only.

TCP is slow to detect connection failures. A client can configure a connection to use a heartbeat to detect connection failure, and can specify a time interval for the heartbeat. If heartbeats are in use, failures will be detected no later than twice the heartbeat interval.

In a Connection URL, heartbeat is set using the `idle_timeout` property, which is an integer corresponding to the heartbeat period in seconds. For instance, the following line from a JNDI properties file sets the heartbeat time out to 3 seconds:

```
connectionfactory.qpidConnectionfactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672',idle_timeout=3
```

Report a bug

9.1.3.2. Failover Behavior and the Qpid Messaging API

The Qpid Messaging API supports automatic reconnection when a connection fails. Senders can be configured to replay any in-doubt messages (messages which were sent but not acknowledged by the broker. See "Connection Options" and "Sender Capacity" in *Messaging Programming Reference* for details.

In C++ and Python clients, heartbeats are disabled by default. You can enable them by specifying a heartbeat interval (in seconds) for the connection using the 'heartbeat' option.

See "Cluster Failover"; in *Messaging Programming Reference* for details on how to keep the client aware of cluster membership.

Report a bug

9.1.4. Error handling

9.1.4.1. Error handling in Clusters

When a broker crashes or is killed, a broker machine fails, a broker connection fails, or a broker hang is detected, the other brokers in the cluster are notified that the failed broker is no longer a member of the cluster.

If a new broker is joined to the cluster, it synchronizes with an active broker to obtain the current cluster state; if this synchronization fails, the new broker exits the cluster and aborts.

If a broker becomes extremely busy and stops responding, it stops accepting incoming work. All other brokers continue processing, and the non-responsive node caches all AIS traffic. When the non-responsive node resumes, it completes processing all cached AIS events, then accepts further incoming work.

Broker hangs are only detected if the watchdog plugin is loaded and the `--watchdog-interval` option is set. The watchdog plug-in kills the qpidd broker process if it becomes stuck for longer than the watchdog interval. In some cases, e.g. certain phases of error resolution, it is possible for a stuck process to hang other cluster members that are waiting for it to send a message. Using the watchdog, the stuck process is terminated and removed from the cluster, allowing other members to continue and clients of the stuck process to fail over to other members.

Redundancy can also be achieved directly in the AIS network by specifying more than one network interface in the AIS configuration file. This causes Totem to use a redundant ring protocol, which makes failure of a single network transparent.

Redundancy can be achieved at the operating system level by using NIC bonding, which

combines multiple network ports into a single group, effectively aggregating the bandwidth of multiple interfaces into a single connection. This provides both network load balancing and fault tolerance.

If any broker encounters an error, the brokers compare notes to see if they all received the same error. If not, the broker removes itself from the cluster and shuts itself down to ensure that all brokers in the cluster have consistent state. For instance, a broker may run out of disk space; if this happens, the broker shuts itself down. Examining the broker's log can help determine the error and suggest ways to prevent it from occurring in the future.

Report a bug

9.1.5. Persistence

9.1.5.1. Persistence in High Availability Messaging Clusters

Persistence and *clustering* are two different ways to provide reliability. Most systems that use a cluster do not enable persistence, but you can do so if you want to ensure that messages are not lost even if the last broker in a cluster fails.

A cluster must have either *all transient* or *all persistent* members: mixed clusters are not allowed.

Each broker in a persistent cluster stores an independent replica of the cluster's state.

Report a bug

9.1.5.2. Clean and Dirty Stores

When a broker is an active member of a cluster, its store is marked "dirty" because it may be out of date compared to other brokers in the cluster. If a broker leaves a running cluster because it is stopped, it crashes or the host crashes, its store continues to be marked "dirty". The store should remain dirty until it belongs to the last broker in the cluster or the cluster is shutdown by the command `qpid-cluster --all-stop`.

If the cluster is reduced to a single broker, its store is marked "clean" since it is the only broker making updates. If the cluster is shut down with the command `qpid-cluster -k` then all the stores are marked clean.

When a cluster is initially formed, brokers with clean stores read from their stores. Brokers with dirty stores, or brokers that join after the cluster is running, discard their old stores and initialize a new store with an update from one of the running brokers. The `--truncate yes` option can be used to force a broker to discard all existing stores even if they are clean. (A dirty store is discarded regardless.)

Discarded stores are copied to a back up directory. The active store is in *<data-dir>*`/rhm`. Back-up stores are in *<data-dir>*`/_cluster.bak.`*<nnnn>*`/rhm`, where *<nnnn>* is a 4 digit number. A higher number means a more recent backup.

Report a bug

9.1.5.3. Starting a Persistent Cluster

When starting a persistent cluster broker, set the `cluster-size` option to the number of brokers in the cluster. This allows the brokers to wait until the entire cluster is running so that they can synchronize their stored state.

The cluster can start if:

- all members have empty stores, or

- at least one member has a clean store

All members of the new cluster will be initialized with the state from a clean store.

Report a bug

9.1.5.4. Stop a Persistent Cluster

To cleanly shut down a persistent cluster use the command `qpid-cluster -k`. This causes all brokers to synchronize their state and mark their stores as "clean" so they can be used when the cluster restarts.

Report a bug

9.1.5.5. Start a Persistent Cluster with no Clean Store

If the cluster has previously had a total failure and there are no clean stores, the brokers will fail to start with the log message `Cannot recover, no clean store.`

When this happens, you can restart the cluster by marking one of the brokers' data directories as "clean". If you can see from time-stamps that one of the data directories is more recent than the other, choose the recent one to mark as clean. However, if all stores are dirty, it means that the brokers all failed simultaneously. In that case, you can choose any broker's data directory.

1. To mark a data directory as clean, look in it for one or more subdirectories in the format of `_cluster.bak.<nnnn>`. These subdirectories will exist if previous restarts were attempted.
   - If no such subdirectory exists, proceed to step 3 of this procedure.
   - If more than one `_cluster.bak.<nnnn>` subdirectory exists, note the one with the highest 4-digit number. It is the newest.
2. Run this sequence of commands on the newest `_cluster.bak.<nnnn>` subdirectory.

```
cd <data-dir>
mv rhm rhm.bak
cp -a _cluster.bak.<nnnn>/rhm .
```

3. Now you are ready to mark this store as clean. Run:

```
qpid-cluster-store -c <data-dir>
```

Now you can restart the cluster, and all brokers' stores will be initialized from the one that you marked as clean.

Report a bug

9.1.5.6. Isolated failures in a Persistent Cluster

A broker in a persistent cluster may encounter errors that other brokers in the cluster do not; if this happens, the broker shuts itself down to avoid making the cluster state inconsistent. For example a disk failure on one node will result in that node shutting down. Running out of storage capacity can also cause a node to shut down because the brokers may not run out of storage at exactly the same point, even if they have similar storage configuration. To avoid unnecessary broker shutdowns, make sure the queue policy size of each durable queue is less than the capacity of the journal for the queue.

Report a bug

9.1.6. Configure a Cluster

9.1.6.1. Configure OpenAIS

1. Clustering is implemented using the `cluster.so` module, which is loaded by default when you start a broker. To run brokers in a cluster, make sure they all use the same OpenAIS `mcastaddr`, `mcastport`, and `bindnetaddr`. All brokers in a cluster must also have the same cluster name. Specify the cluster name in `qpidd.conf`:

```
cluster-name="loctest_cluster"
```

Note

Cluster names need to be 15 characters or less. Using a longer name may not be handled gracefully.

2. a. On Red Hat Enterprise Linux 6, create the file `/etc/corosync/uidgid.d/qpidd` to tell Corosync the name of the user running the broker. By default, the user is qpidd:

```
uidgid {
   uid: qpidd
   gid: qpidd
}
```

   b. On Red Hat Enterprise Linux 5, the primary group for the process running qpidd must be the ais group. If you are running qpidd as a service, it is run as the `qpidd` user, which is already in the ais group. If you are running the broker from the command line, ensure that the primary group for the user running qpidd is ais. You can set the primary group using `newgrp`:

```
$ newgrp ais
```

Report a bug

9.1.6.2. Firewall Configuration for Clustering

The following ports are used on a clustered system, and must be opened on the firewall:

Table 9.1. Ports Used by Clustered Systems

Port	Protocol	Component
5404	UDP	cman
5405	UDP	cman
5405	TCP	luci
8084	TCP	luci
11111	TCP	ricci
14567	TCP	gnbd
16851	TCP	modclusterd
21064	TCP	dlm
50006	TCP	ccsd
50007	UDP	ccsd
50008	TCP	ccsd
50009	TCP	ccsd

The following `iptables` commands, when run with root privileges, will configure the system to

allow communication on these ports.

```
iptables -I INPUT -p udp -m udp --dport 5405  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 5405  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 8084  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 11111  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 14567  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 16851  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 21064  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 50006  -j ACCEPT
iptables -I INPUT -p udp -m udp --dport 50007  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 50008  -j ACCEPT
iptables -I INPUT -p tcp -m tcp --dport 50009  -j ACCEPT
service iptables save
service iptables restart
```

Report a bug

9.1.6.3. Configure a Broker to run in a Messaging Cluster

Procedure 9.1. Configure a Broker to run in a Messaging Cluster

1. Use `ifconfig` to find the inet addr and the netmask for the interface you want:

```
# ifconfig
eth0  Link encap:Ethernet  HWaddr 00:E0:81:76:B6:C6
      inet addr:10.16.44.222  Bcast:10.16.47.255  Mask:255.255.248.0
      inet6 addr: fe80::2e0:81ff:fe76:b6c6/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:35914541 errors:6 dropped:0 overruns:0 frame:6
      TX packets:6529841 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:20294124383 (18.9 GiB)  TX bytes:12925473031 (12.0 GiB)
      Interrupt:98 Base address:0x8000
```

2. The binding address in `/etc/ais/openais.conf` should be the network address for the interface, which you can find by doing a bitwise `AND` of the inet addr (in this case, 10.16.44.222) and the network mask (in this case, 255.255.248.0). The result is 10.16.40.0. As a sanity check, you can use `route` and make sure the address you computed is associated with the interface:

```
$ /sbin/route
Kernel IP routing table
Destination    Gateway        Genmask         Flags Metric Ref Use Iface
20.0.10.0      *               255.255.255.0   U     0      0      0 eth1
192.168.122.0 *               255.255.255.0   U     0      0      0 virbr0
10.16.40.0     *               255.255.248.0   U     0      0      0 eth0
169.254.0.0    *               255.255.0.0     U     0      0      0 eth1
default        10.16.47.254   0.0.0.0         UG    0      0      0 eth0
```

3. To use `eth0` as the interface for the cluster, find the setting for bindnetaddr in `/etc/ais/openais.conf`, and set it to 10.16.40.0:

```
bindnetaddr: 10.16.40.0
```

4. Make sure that the primary group for the user running qpidd is “ais”. For instance, if you are running qpidd as a daemon, the user is named `qpidd`. Make `ais` the primary group for `qpidd` as follows:

```
# usermod -g ais qpidd
```

5. Set the name of the cluster in qpidd.conf.

```
cluster-name="Mick"
```

6. After launching the cluster members, use `qpid-cluster` to list the membership of the cluster or to cleanly shut down the cluster members.

```
[conway@mrg32 ~]$ qpid-cluster mrg33
  Cluster Name: Mick
Cluster Status: ACTIVE
  Cluster Size: 3
       Members: ID=20.0.100.33:22689
URL=amqp:tcp:20.0.10.33:5672,tcp:10.16.44.238:5672,\
tcp:20.0.100.33:5672,tcp:192.168.122.1:5672
               : ID=20.0.100.34:20810
URL=amqp:tcp:20.0.10.34:5672,tcp:10.16.44.239:5672,\
tcp:20.0.100.34:5672,tcp:192.168.122.1:5672
               : ID=20.0.100.35:20139
URL=amqp:tcp:20.0.10.35:5672,tcp:20.0.20.35:5672,tc\
p:10.16.44.240:5672,tcp:20.0.100.35:5672,tcp:192.168.122.1:5672
```

7. You can also run `qpid-tool` against any cluster node to view details of the cluster. The cluster is one of the objects shown by the `list` command.

```
qpid: list
Management Object Types:
ObjectType                               Active  Deleted
=================================================
com.redhat.rhm.store:journal        1        0
com.redhat.rhm.store:store          1        0
org.apache.qpid.broker:binding      5        0
org.apache.qpid.broker:broker       1        0
org.apache.qpid.broker:connection   1        0
org.apache.qpid.broker:exchange     7        0
org.apache.qpid.broker:queue        2        0
org.apache.qpid.broker:session      1        0
org.apache.qpid.broker:system       1        0
org.apache.qpid.broker:vhost        1        0
org.apache.qpid.cluster:cluster     1        0
```

8. To see the properties of the cluster, use `show cluster`:

```
qpid: show cluster
Object of type org.apache.qpid.cluster:cluster: (last sample time: 13:56:40)
Type      Element       112
================================================================================
========================
property  brokerRef     102
property  clusterName   foo
property  clusterID     da821ff9-2a88-4002-b976-f18680556290
property  publishedURL
amqp:tcp:10.16.44.222:52265,tcp:20.0.10.15:52265,tcp:192.168.122.1:52265
property  clusterSize   1
property  status        ACTIVE
property  members
amqp:tcp:10.16.44.222:52265,tcp:20.0.10.15:52265,tcp:192.168.122.1:52265
```

Report a bug

9.1.6.4. Start a Broker in a Messaging Cluster

- After you have configured OpenAIS, run the broker from the command line. Specify the cluster name.

```
[example@localhost]$ qpidd --cluster-name="loctest_cluster"
```

All brokers in a cluster must have identical configuration, as listed:

- The brokers must load the same set of plug-ins.
- The brokers must have matching configuration files and command line arguments.
- The brokers must have identical ACL files and SASL databases.
- If one broker uses persistence, all must use persistence. A mix of transient and persistent brokers is not allowed.

Differences in configuration can cause brokers to exit the cluster. For instance, if different ACL settings allow a client to access a queue on broker A but not on broker B, then publishing to the queue will succeed on A and fail on B, so B will exit the cluster to prevent inconsistency.

However, there are a number of cluster configuration options which do not need to be identical. The following settings can differ for brokers on a given cluster:

- logging options
- cluster-url - if set, it will be different for each broker.
- port - brokers can listen on different ports.

Note

If you are using SELinux, the qpidd process and OpenAIS must have the same SELinux context, or else SELinux must be set to permissive mode. If both qpidd and OpenAIS are run as services, they have the same SELinux context. If both OpenAIS and qpidd are run as user processes, they have the same SELinux context. If one is run as a service, and the other is run as a user process, they have different SELinux contexts.

Report a bug

9.1.6.5. Clustering options

Table 9.2. Options for High Availability Messaging Clusters

Option	Description
`--cluster-name` *NAME*	Name of the Messaging Cluster to join. A Messaging Cluster consists of all brokers started with the same cluster-name and OpenAIS configuration.
`--cluster-size` *N*	Wait for at least N initial members before completing cluster initialization and serving clients. Use this option in a persistent cluster so all brokers in a persistent cluster can exchange the status of their persistent store and do consistency checks before serving clients.
`--cluster-url` *URL*	An AMQP URL containing the local address that the broker advertises to clients for fail-over connections. This is different for each host. By default, all local addresses for the broker are advertised. You only need to set this if 1. Your host has more than one active network interface, and 2. You want to restrict client fail-over to a specific interface or interfaces. Each broker in the cluster is specified using the form: `url = ["amqp:"][ user ["/" password] "@" ] protocol_addr *("," protocol_addr)` `protocol_addr = tcp_addr / rmda_addr / ssl_addr / ...` `tcp_addr = ["tcp:"] host [":" port]` `rdma_addr = "rdma:" host [":" port]` `ssl_addr = "ssl:" host [":" port]` In most cases, only one address is advertised, but more than one address can be specified in the machine running the broker has more than one network interface card, and you want to allow clients to connect using multiple network interfaces. Use a comma delimiter (",") to separate brokers in the URL. Examples: - `amqp:tcp:192.168.1.103:5672` advertises a single address to the broker for failover. - `amqp:tcp:192.168.1.103:5672, tcp:192.168.1.105:5672` advertises two different addresses to the broker for failover, on two different network interfaces.
`--cluster-cman`	CMAN protects against the "split-brain" condition, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. When "split-brain" occurs, each of the sub-clusters can access shared resources without knowledge of the other sub-cluster, resulting in corrupted cluster integrity.

	To avoid "split-brain", CMAN uses the notion of a "quorum". If more than half the cluster nodes are active, the cluster has quorum and can act. If half (or fewer) nodes are active, the cluster does not have quorum, and all cluster activity is stopped. There are other ways to define the quorum for particular use cases (e.g. a cluster of only 2 members), see the CMAN documentation for more detail. When enabled, the MRG broker will wait until it belongs to a quorate cluster before accepting client connections. It continually monitors the quorum status and shuts down immediately if the node it runs on loses touch with the quorum.
--cluster-username	SASL username for connections between brokers.
--cluster-password	SASL password for connections between brokers.
--cluster-mechanism	SASL authentication mechanism for connections between brokers

Report a bug

9.1.7. Avoiding Race Conditions in Clusters

9.1.7.1. Race Conditions in Clusters

Multi-program race conditions can arise in multi-broker Qpid messaging systems. Occasional unexpected behaviors may occur, caused by the fact that information takes a finite amount of time to propagate across the brokers in a cluster.

When building a clustered messaging system, it is important to understand the impact of latencies introduced by cluster replication, and take appropriate measures.

Report a bug

9.1.7.2. General Considerations

1. There is some latency in syncing brokers in a cluster.
2. Adding or removing brokers, federated links, etc. to a broker takes time to propagate across the cluster to reach a consistent cluster state.
3. The timing of controlled broker shutdowns is critical. When making changes to broker and federation topologies, allow time for syncing before administrating shutdown procedures of any broker in the cluster.

Report a bug

9.1.7.3. Persistent cluster and "no clean store"

A cluster that uses the message store for persistence has one copy of the store for each cluster broker. When the cluster is shut down, by accident or design, the broker needs to determine which copy of the store should be used when the cluster is restarted.

All of the brokers in the cluster need to start up with identical stores, and they need to use the

store that was owned by the Last Man Standing from the previous instance of the cluster.

But when a cluster is being shut down, it takes a finite amount of time for the last remaining broker to notice that he is indeed the Last Man Standing, and to mark his store as the "clean" one. It is posssible for a test script to kill all of the cluster's brokers so quickly that the last survivor does not have sufficient time to mark his store. In that case, when you attempt to restart the cluster, you will see a "`no clean store`" error message, and the cluster will refuse to start.

You will then have to manually mark one of the stores as clean.

Recommended Practice

To shut down the cluster safely, use:

```
qpid-cluster --all-stop
```

It will perform a coordinated shutdown that will leave all stores clean.

See Also:

- Section 9.1.5.5, “Start a Persistent Cluster with no Clean Store”

Report a bug

9.1.7.4. Federation-of-Clusters Topology Change

Suppose that you have two clusters, `A` and `B`, each comprised of two brokers, `1` and `2`, and you want to federate the two clusters. To federate them, you will add a route whose source broker is `B1` and whose destination broker is `A1`, using this command:

```
qpid-route -s route add   \
${HOST_A_1}:${PORT_A_1} \
${HOST_B_1}:${PORT_B_1} \
${EXCHANGE}           \
${KEY}                \
-d
```

The `-d` makes the route durable, so that it will be restored if cluster `B` is shutdown and then restarted.

But note that this topology change has so far been communicated only to the #1 brokers in both clusters. Information about the change will take a small amount of time to propagate to the #2 brokers in both clusters. The amount of time required will vary, depending on system load.

And now suppose that your script decides to kill broker `B1` first -- before it has been able to communicate the topology change to `B2`. This means that broker B2 will now be the Last Man Standing in cluster `B` -- and its store contains no knowledge of your route! When you restart cluster `B`, the route will not be restored.

Recommended Practice

Before shutting down the brokers in cluster `B`, use these commands:

```
qpid-config exchanges --bindings --broker-addr=${HOST_B_1}:${PORT_B_1}
qpid-config exchanges --bindings --broker-addr=${HOST_B_2}:${PORT_B_2}
```

Use the output to confirm that your route is known to both brokers of the cluster before shutting down either.

Report a bug

9.1.7.5. Newbie Broker Update

When a new broker is added to a cluster, it gets updated to the current cluster state with this process:

1. The newbie broadcasts an update request
2. Veteran brokers make update offers to it.
3. The newbie chooses one.
4. The chosen veteran sends the newbie all state-update information

This process will take a variable amount of time, depending on cluster load. If it is interrupted by killing the veteran broker before the update is complete, the newbie will also exit. If there were only two brokers in your cluster, you no longer have a cluster!

Recommended Practice

1. When a client tries to connect to a clustered broker that is not yet updated, it will block until the broker is ready. You can use this behavior to determine when the newbie update has completed. When your client is able to connect, the newbie update is complete.
2. If the log level on the newbie broker is set to debug or greater the newbie broker will output a line to its log that contains the string "`update completed`".

Report a bug

9.1.8. Troubleshooting

9.1.8.1. Troubleshooting Cluster configuration

If a broker is unable to establish a connection to another broker in the cluster, the log will contain SASL errors, for example:

```
2012-aug-04 10:17:37 info SASL: Authentication failed: SASL(-13): user not found:
Password verification failed
```

Procedure 9.2. Troubleshooting Cluster Configuration

1. Set the SASL user name and password used to connect to other brokers using the `cluster-username` and `cluster-password` properties when you start the broker.
2. Set the SASL mode using `cluster-mechanism`. Any mechanism you enable for broker-to-broker communication can also be used by a client, so do not enable `cluster-mechanism=ANONYMOUS` in a secure environment.
3. Once the cluster is running, run `qpid-cluster` to make sure that the brokers are running as one cluster.
4. If the cluster is correctly configured, queues and messages are replicated to all brokers in the cluster. To test the cluster, run a program that routes messages to a queue on one broker, then connect to a different broker in the same cluster and read the messages to make sure they have been replicated. Use the `drain` and `spout` programs for this test.

Report a bug

9.2. Cluster management

9.2.1. Cluster Management using qpid-cluster

`qpid-cluster` is a command-line utility that allows you to view information on a cluster and its brokers, disconnect a client connection, shut down a broker in a cluster, or shut down the entire

cluster. You can see the options using the `--help` option:

```
$ qpid-cluster --help
```

Report a bug

9.3. Queue Replication for Disaster Recovery

9.3.1. Queue Replication

A *replicated queue* is a queue that generates notification events when a message is enqueued or dequeued, so that its state can be replicated. It is a queue on a source broker that is replicated to the backup queue on a backup broker. A queue is declared to be a replicated queue when it is first created.

A *backup queue* is a queue on a *backup broker* that replicates the state of a replicated queue on a *source broker*, so that applications can access the backup queue if the replicated queue becomes unavailable. A backup queue always has the same name as the corresponding replicated queue.

Queue replication is done by creating a message queue called a *replication event queue* or *replication queue* on the source broker, and subscribing the backup broker to the replication event queue. The source broker sends a message to the replication event queue for each enqueue or dequeue event on the replicated queue. When the backup broker receives an enqueue or dequeue event, it modifies the backup queue accordingly.

The replication mechanism will behave as a load balancer when more than one backup broker is configured for a replicated queue. Unlike High Availability Messaging Clusters, queue replication does not slow broker performance, and works well across a Wide Area Network.

Report a bug

9.3.2. Configuring Queue Replication

9.3.2.1. Source Broker

Queue replication requires configuration on both the source broker and the backup broker. In addition, a federated message route must be created between the replication event queue on the source broker and the replication exchange on the destination broker. To configure queue replication on the source broker:

- Enable a replication plugin
- Start the Source Broker
- Specify a replication event queue when the broker starts
- Create one or more replicating queues

Report a bug

9.3.2.2. Backup Broker

Queue replication requires configuration on both the source broker and the backup broker. In addition, a federated message route must be created between the replication event queue on the source broker and the replication exchange on the destination broker. To configure queue replication on the backup broker:

- Ensure that the replication exchange plugin is configured
- Start the backup broker

- Create a replication exchange
- Create a backup queue that corresponds to each replicated queue

Report a bug

9.3.2.3. Configure Queue Replication on the Source Broker Changes

To configure queue replication on the source broker:

1. Specify a replication event queue when the broker starts
2. Create one or more replicating queues.

Queue replication uses the replication event plugin. This plugin is automatically loaded when present on the system. The plugin file is `/usr/lib64/qpid/daemon/replication_exchange.so` for 64-bit systems, or `/usr/lib/qpid/daemon/replication_exchange.so` for 32-bit systems.

Procedure 9.3. Configure Queue Replication on the Source Broker

1. **Specify replication event queue at broker start-up**
   A. To use an existing durable queue as the replication event queue, start the broker with the `--replication-queue` option, giving the name of the existing queue as an argument.
   B. To create a new transient queue for the replication event queue, start the broker with the `--create-replication-queue` option, giving the name for the new queue as the argument.
   C. To configure the broker via `qpidd.conf`, for use when the broker starts as a service, add the following lines:

   ```
   replication-queue=my_repl_event_queue
   create-replication-queue=true
   ```

   `Specify the name of the queue to use as the replication event queue where` *`my_repl_event_queue`* appears. If `create-replication-queue` is set to `true`, a transient queue is created. If set to `false`, an queue with that name must exist.
2. **Restart the Message Broker**
   As the root user, issue the command:

   ```
   service qpidd restart
   ```
3. **Create one or more replicated queues**
   A. The `qpid-config` command can be used to create queues. The `--generate-queue-events` option generates replication events. Specify its value as `1` to replicate enqueues. Specify its value as `2` to replicates both enqueues and dequeues.

   For example:

   ```
   qpid-config --broker-addr src-host add queue a-replicated-queue --
   generate-queue-events 2
   ```

   The previous command creates a queue called `a-replicated-queue` that replicates both enqueues and dequeues.
   B. If a queue is created programatically, the key `qpid.queue_event_generation` enables replication. Set to `1` to replicate enqueue events; set to `2` to replicate both enqueue and dequeue events.

   For example, in C++ this can be done with the following code:

   C++

```
QueueOptions options;
options.setInt("qpid.queue_event_generation", 2); // both enqueue and
dequeue
session.queueDeclare(arg::queue="a-replicated-queue",
arg::arguments=options);
```

Report a bug

9.3.2.4. Configure Queue Replication on the Backup Broker

To configure queue replication on the backup broker:

1. Create a replication exchange.
2. Create a backup queue for each replicating queue.

Queue replication uses the replicating listener plugin. This plugin is automatically loaded when present on the system. The plugin file is `/usr/lib64/qpid/daemon/replicating_listener.so` for 64-bit systems, or `/usr/lib/qpid/daemon/replicating_listener.so` for 32-bit systems.

Procedure 9.4. Configure Queue Replication on the Backup Broker

1. **Create a Replication Exchange**
   With the backup broker running, use `qpid-config` to create a replication exchange:

   ```
   qpid-config --broker-addr backup-host add exchange replication replication-
   exchange
   ```

   Where: *`backup-host`* is the address of the backup broker; and *`replication-exchange`* is the arbitrary name that you give to the replication exchange.
2. **Create a backup queue for each replicating queue**
   As the root user, issue the command:

   ```
   service qpidd restart
   ```
3. **Create a backup queue for each replicated queue**
   The `qpid-config` can be used to create queues. For each replicated queue, create one backup queue with the same name as the replicated queue:

   ```
   qpid-config --broker-addr backup-host add queue my_repl_queue
   ```

   Where: *`backup-host`* is the address of the backup broker; and *`my_repl_queue`* is the name of the replicated queue that this queue will back up.

Report a bug

9.3.2.5. Create Message Route from Source Broker to Backup Broker

Queue replication requires a federated message route between the replication event queue on the source broker and the replication exchange on the backup broker.

The following example `qpid-config` command creates a federated message route between the *`my_repl_event_queue`* replication queue on the source broker with network address *`src-host`* and the replication exchange *`replication-exchange`* on the backup broker with network address *`backup-host`*.

Additionally, it enables acknowledgment to ensure that replication events are not lost if the route fails, configuring a batch size of 50 for acknowledgments for efficiency.

```
qpid-route --ack 50 queue add backup-host src-host replication-exchange
my_repl_event_queue
```

When the source broker replication event queue, source broker replicating queues, backup broker replication exchange, backup broker backup queues, and federated message route are all created, replication functions when the brokers are started.

Report a bug

9.3.2.6. Queue Replication Example

In this example, the source broker is on host1 and the backup is on host2.

On host1 we start the source broker and specifcy that a queue called 'replication' should be used for storing the events until consumed by the backup. We also request that this queue be created (as transient) if not already specified:

Add the following lines to `/etc/qpidd.conf`:

```
replication-queue=replication-queue
create-replication-queue=true
log-enable=info+
```

Now start the broker:

```
service qpidd start
```

On host2 we start up the backup broker:

```
service qpidd start
```

We can then create the instance of that replication exchange that we will use to process the events:

```
qpid-config -a host2 add exchange replication replication-exchange
```

We then connect the replication queue on the source broker with the replication exchange on the backup broker using the qpid-route command:

```
qpid-route --ack 50 queue add host2 host1 replication-exchange replication-queue
```

The example above configures the bridge to acknowledge messages in batches of 50.

Now create two queues (on both source and backup brokers), one replicating both enqueues and dequeues (`queue-a`) and the other replicating only dequeues (`queue-b`):

```
qpid-config -a host1 add queue queue-a --generate-queue-events 2
qpid-config -a host1 add queue queue-b --generate-queue-events 1

qpid-config -a host2 add queue queue-a
qpid-config -a host2 add queue queue-b
```

We are now ready to use the queues and see the replication.

Any message enqueued on `queue-a` will be replicated to the backup broker. When the message is acknowledged by a client connected to host1 (and thus dequeued), that message will be removed from the copy of the queue on host2. The state of `queue-a` on host2 will thus mirror that of the equivalent queue on host1, albeit with a small lag. (Note however that we must not have clients connected to host2 publish to - or consume from - `queue-a` or the state will fail to replicate correctly due to conflicts).

Any message enqueued on queue-b on host1 will also be enqueued on the equivalent queue on host2. However the acknowledgement and consequent dequeuing of messages from queue-b on host1 will have no effect on the state of queue-b on host2.

Report a bug

9.3.3. Using Backup Queues in Messaging Clients

9.3.3.1. Concept: Using Backup Queues in Messaging Clients

If a messaging client connects to a backup queue and performs operations that change the state of the queue, it will no longer represent the state of the replicated queue. If a client is expected to read and remove messages from the backup queue, then message enqueues should be replicated, but not message dequeues. Browsing a queue does not cause problems with replication.

Report a bug

9.3.4. Queue Replication and High Availability

9.3.4.1. Queue Replication and High Availability

The source or backup broker can be in a cluster. If either broker is in a cluster and fails, the federated bridge is re-established using another broker in the same cluster.

The members of each cluster are communicated when the messaging route is established. If the source broker becomes unavailable, the backup cluster can fail over to another broker from the source broker's cluster, but only to a broker that was in the cluster at the time the messaging route was created. If new brokers have been added to the source broker's cluster, removing the messaging route and creating a new one makes the new brokers available for failover.

New brokers added to a backup cluster do not automatically receive information about established message routes. Removing the messaging route and creating a new one makes these routes available to all brokers in the backup cluster.

If acknowledgments are enabled for the messaging route, deleting the route and creating a new one does not result in loss of messages.

Report a bug

Chapter 10. Broker Federation

10.1. Broker Federation

Broker Federation allows messaging networks to be defined by creating *message routes*, in which messages in one broker (the *source broker*) are automatically routed to another broker (the *destination broker*). These routes can be defined between exchanges in the two brokers (the *source exchange* and the *destination exchange*), or from a queue in the source broker (the *source queue*) to an exchange in the destination broker.

Message routes are unidirectional; when bidirectional flow is needed, one route is created in each direction. Routes can be durable or transient. A durable route survives broker restarts, restoring a route as soon as both the source broker and the destination are available. If the connection to a destination is lost, messages associated with a durable route continue to accumulate on the source, so they can be retrieved when the connection is reestablished.

Broker Federation can be used to build large messaging networks, with many brokers, one route at a time. If network connectivity permits, an entire distributed messaging network can be configured from a single location. The rules used for routing can be changed dynamically as servers or responsibilities change at different times of day, or to reflect other changing conditions.

Report a bug

10.2. Broker Federation Use Cases

Broker Federation is useful in a wide variety of scenarios. Some of these have to do with functional organization; for instance, brokers can be organized by geography, service type, or priority. Here are some use cases for federation:

- Geography: Customer requests can be routed to a processing location close to the customer.
- Service Type: High value customers can be routed to more responsive servers.
- Load balancing: Routing among brokers can be changed dynamically to account for changes in actual or anticipated load.
- High Availability: Routing can be changed to a new broker if an existing broker becomes unavailable.
- WAN Connectivity: Federated routes can connect disparate locations across a wide area network, while clients connect to brokers on their own local area network. Each broker can provide persistent queues that can hold messages even if there are gaps in WAN connectivity.
- Functional Organization: The flow of messages among software subsystems can be configured to mirror the logical structure of a distributed application.
- Replicated Exchanges: High-function exchanges like the XML exchange can be replicated to scale performance.
- Interdepartmental Workflow: The flow of messages among brokers can be configured to mirror interdepartmental workflow at an organization.

Report a bug

10.3. Broker Federation Overview

10.3.1. Message Routes

Broker Federation is done by creating message routes. The destination for a route is always an exchange on the destination broker.

By default, a message route is created by configuring the destination broker, which then contacts the source broker to subscribe to the source queue. This is called a *pull route*.

It is also possible to create a route by configuring the source broker, which then contacts the destination broker to send messages. This is called a *push route*, and is particularly useful when the destination broker may not be available at the time the messaging route is configured, or when a large number of routes are created with the same destination exchange.

The source for a route can be either an exchange or a queue on the source broker. If a route is between two exchanges, the routing criteria can be given explicitly, or the bindings of the destination exchange can be used to determine the routing criteria. To support this functionality, there are three kinds of message routes:

- Queue routes
- Exchange routes
- Dynamic exchange routes

Report a bug

10.3.2. Queue Routes

Queue Routes route all messages from a source queue to a destination exchange. If message acknowledgment is enabled, messages are removed from the queue when they have been received by the destination exchange; if message acknowledgment is off, messages are removed from the queue when sent.

When there are multiple subscriptions on an AMQP queue, messages are distributed among subscribers. For example, two queue routes being fed from the same queue will each receive a load-balanced number of messages.

If fanout behavior is required instead of load-balancing, use an *exchange route*.

Report a bug

10.3.3. Exchange Routes

Exchange routes route messages from a source exchange to a destination exchange, using a binding key (which is optional for a fanout exchange).

Internally, creating an exchange route creates a private queue (auto-delete, exclusive) on the source broker to hold messages that are to be routed to the destination broker, binds this private queue to the source broker exchange, and subscribes the destination broker to the queue.

Report a bug

10.3.4. Dynamic Exchange Routes

A *dynamic exchange route* connects an exchange on one broker (the source) to an exchange on another broker (the destination) so that a client can subscribe to the exchange on the destination broker, and receive messages that match that subscription sent to either exchange.

Dynamic exchange routes are *directional*. When a dynamic exchange route is created from broker A to broker B, broker B dynamically creates and deletes subscriptions to broker A's exchange on behalf of its clients. In this way broker B acts as a proxy for its clients. When a client subscribes to the destination exchange on broker B, broker B subscribes the exchange to broker A's source exchange with the subscription that the client created. Clients subscribing to broker B effectively subscribe to both the exchange on broker B and the dynamically routed exchange on broker A.

The consuming broker creates and deletes subscriptions to the source broker's dynamically routed exchange as clients subscribe and unsubscribe to the destination exchange on the consuming broker. It is important to note that the dynamic exchange route is a competing subscription, so the destination broker is a message consumer like any other subscriber.

In a dynamic exchange route, the source and destination exchanges must have the same exchange type, and they must have the same name. For instance, if the source exchange is a direct exchange, the destination exchange must also be a direct exchange, and the names must match.

Internally, dynamic exchange routes are implemented in the same way as exchange routes, except that the bindings used to implement dynamic exchange routes are modified if the bindings in the destination exchange change.

A dynamic exchange route is always a pull route. It can never be a push route.

See Also:

- Section 10.4.7, “Create and Delete Dynamic Exchange Routes”

Report a bug

10.3.5. Federation Topologies

A federated network is generally a tree, star, or line, using bidirectional links (implemented as a pair of unidirectional links) between any two brokers. A ring topology is also possible, if only unidirectional links are used.

Every message transfer takes time. For better performance, minimize the number of brokers between the message origin and final destination. In most cases, tree or star topologies do this best.

For any pair of nodes A, B in a federated network, if there is more than one path between A and B, ensure that it is not possible for any messages to cycle between A and B. Looping message traffic can flood the federated network. The tree, star and line topologies do not have message loops. A ring topology with bidirectional links is one example of a topology that causes this problem, because a given broker can receive the same message from two different brokers. Mesh topologies can also cause this problem.

Report a bug

10.3.6. Federation Among High Availability Clusters

Federation is generally used together with High Availability Message Clusters, using clusters to provide high availability on each LAN, and federation to route messages among the clusters.

To create a message route between two clusters, create a route between any one broker in the first cluster and any one broker in the second cluster. Each broker in a given cluster can use message routes defined for another broker in the same cluster. If the broker for which a message route is defined should fail, another broker in the same cluster can restore the message route.

Report a bug

10.4. Configuring Broker Federation

10.4.1. The qpid-route Utility

`qpid-route` is a command line utility used to configure federated networks of brokers and to view the status and topology of networks. It can be used to configure routes among any brokers that `qpid-route` can connect to.

Report a bug

10.4.2. qpid-route Syntax

The syntax of `qpid-route` is as follows:

```
qpid-route [OPTIONS] dynamic add <dest-broker> <src-broker> <exchange>
qpid-route [OPTIONS] dynamic del <dest-broker> <src-broker> <exchange>

qpid-route [OPTIONS] route add <dest-broker> <src-broker> <exchange> <routing-key>
qpid-route [OPTIONS] route del <dest-broker> <src-broker> <exchange> <routing-key>

qpid-route [OPTIONS] queue add <dest-broker> <src-broker> <dest-exchange>  <src-
queue>
qpid-route [OPTIONS] queue del <dest-broker> <src-broker> <dest-exchange>  <src-
queue>

qpid-route [OPTIONS] route list  [<broker>]
qpid-route [OPTIONS] route flush [<broker>]
qpid-route [OPTIONS] route map   [<broker>]

qpid-route [OPTIONS] link add  <dest-broker> <src-broker>
qpid-route [OPTIONS] link del  <dest-broker> <src-broker>
qpid-route [OPTIONS] link list [<dest-broker>]
```

The syntax for `broker`, `dest-broker`, and `src-broker` is as follows:

```
[username/password@] hostname | ip-address [:<port>]
```

The following are all valid examples of the above syntax: `localhost`, `10.1.1.7:10000`, `broker-host:10000`, `guest/guest@localhost`.

Report a bug

10.4.3. qpid-route Options

Table 10.1. `qpid-route` **options to manage federation**

Option	Description
`-v`	Verbose output.
`-q`	Quiet output, will not print duplicate warnings.
`-d`	Make the route durable.
`-e`	Delete link after deleting the last route on the link.
`--timeout N`	Maximum time to wait when qpid-route connects to a broker, in seconds. Default is 10 seconds.
`--ack N`	Acknowledge transfers of routed messages in batches of N. Default is 0 (no acknowledgments). Setting to 1 or greater enables acknowledgments; when using acknowledgments, values of N greater than 1 can significantly improve performance, especially if there is significant network latency between the two brokers.
`-s [ --src-local ]`	Configure the route in the source broker (create a push route).
`-t <transport> [ --transport <transport>]`	Transport protocol to be used for the route. • tcp (default) • ssl • rdma
`--client-sasl-mechanism <mech>`	SASL mechanism for authentication when the client connects to the destination broker (for example: EXTERNAL, ANONYMOUS, PLAIN, CRAM-MD, DIGEST-MD5, GSSAPI).

Report a bug

10.4.4. Create and Delete Queue Routes

1. To create and delete queue routes, use the following syntax:

```
qpid-route [OPTIONS] queue add <dest-broker> <src-broker> <dest-exchange>
<src-queue>
qpid-route [OPTIONS] queue del <dest-broker> <src-broker> <dest-exchange>
<src-queue>
```

2. For example, use the following command to create a queue route that routes all messages from the queue named `public` on the source broker `localhost:10002` to the `amq.fanout` exchange on the destination broker `localhost:10001`:

```
$ qpid-route queue add localhost:10001 localhost:10002 amq.fanout public
```

3. Optionally, specify the `-d` option to persist the queue route. The queue route will be restored if one or both of the brokers is restarted:

```
$ qpid-route -d queue add localhost:10001 localhost:10002 amq.fanout public
```

4. The `del` command takes the same arguments as the `add` command. Use the following

command to delete the queue route described above:

```
$ qpid-route queue del localhost:10001 localhost:10002 amq.fanout public
```

Report a bug

10.4.5. Create and Delete Exchange Routes

1. To create and delete exchange routes, use the following syntax:

```
qpid-route [OPTIONS] route add <dest-broker> <src-broker> <exchange>
<routing-key>
qpid-route [OPTIONS] route del <dest-broker> <src-broker> <exchange>
<routing-key>
qpid-route [OPTIONS] flush [<broker>]
```

2. For example, use the following command to create an exchange route that routes messages that match the binding key `global.#` from the `amq.topic` exchange on the source broker `localhost:10002` to the `amq.topic` exchange on the destination broker `localhost:10001`:

```
$ qpid-route route add localhost:10001 localhost:10002 amq.topic global.#
```

3. In many applications, messages published to the destination exchange must also be routed to the source exchange. Create a second exchange route, reversing the roles of the two exchanges:

```
$ qpid-route route add localhost:10002 localhost:10001 amq.topic global.#
```

4. Specify the `-d` option to persist the exchange route. The exchange route will be restored if one or both of the brokers is restarted:

```
$ qpid-route -d route add localhost:10001 localhost:10002 amq.fanout public
```

5. The `del` command takes the same arguments as the `add` command. Use the following command to delete the first exchange route described above:

```
$ qpid-route route del localhost:10001 localhost:10002 amq.topic global.#
```

Report a bug

10.4.6. Delete All Routes for a Broker

- Use the `flush` command to delete all routes for a given broker:

```
qpid-route [OPTIONS] route flush [<broker>]
```

For instance, use the following command to delete all routes for the broker `localhost:10001`:

```
$ qpid-route route flush localhost:10001
```

Report a bug

10.4.7. Create and Delete Dynamic Exchange Routes

1. To create and delete dynamic exchange routes, use the following syntax:

```
qpid-route [OPTIONS] dynamic add <dest-broker> <src-broker> <exchange>
qpid-route [OPTIONS] dynamic del <dest-broker> <src-broker> <exchange>
```

2. Create a new topic exchange on each of two brokers:

```
$ qpid-config -a localhost:10003 add exchange topic fed.topic
$ qpid-config -a localhost:10004 add exchange topic fed.topic
```

3. Create a dynamic exchange route that routes messages from the `fed.topic` exchange on the source broker `localhost:10004` to the `fed.topic` exchange on the destination broker `localhost:10003`:

```
$ qpid-route dynamic add localhost:10003 localhost:10004 fed.topic
```

Internally, this creates a private autodelete queue on the source broker, and binds that queue to the `fed.topic` exchange on the source broker, using each binding associated with the `fed.topic` exchange on the destination broker.

4. In many applications, messages published to the destination exchange must also be routed to the source exchange. Create a second dynamic exchange route, reversing the roles of the two exchanges:

```
$ qpid-route dynamic add localhost:10004 localhost:10003 fed.topic
```

5. Specify the `-d` option to persist the exchange route. The exchange route will be restored if one or both of the brokers is restarted:

```
$ qpid-route -d dynamic add localhost:10004 localhost:10003 fed.topic
```

When an exchange route is durable, the private queue used to store messages for the route on the source exchange is also durable. If the connection between the brokers is lost, messages for the destination exchange continue to accumulate until it can be restored.

6. The `del` command takes the same arguments as the `add` command. Delete the first exchange route described above:

```
$ qpid-route dynamic del localhost:10004 localhost:10003 fed.topic
```

Internally, this deletes the bindings on the source exchange for the private queues associated with the message route.

Report a bug

10.4.8. View Routes

Procedure 10.1. Using the `route list` command

1. Create the following two routes:

```
$ qpid-route dynamic add localhost:10003 localhost:10004 fed.topic
$ qpid-route dynamic add localhost:10004 localhost:10003 fed.topic
```

2. Use the `route list` command to show the routes associated with the broker:

```
$ qpid-route route list localhost:10003
localhost:10003 localhost:10004 fed.topic <dynamic>
```

Note that this shows only one of the two routes created, namely the route for which `localhost:10003` is a destination.

3. To view the route for which `localhost:10004` is a destination, run `route list` on `localhost:10004`:

```
$ qpid-route route list localhost:10004
localhost:10004 localhost:10003 fed.topic <dynamic>
```

Procedure 10.2. Using the `route map` command

1. The `route map` command shows all routes associated with a broker, and recursively displays all routes for brokers involved in federation relationships with the given broker. For example, run the `route map` command for the two brokers configured above:

```
$ qpid-route route map localhost:10003

Finding Linked Brokers:
    localhost:10003... Ok
    localhost:10004... Ok

Dynamic Routes:

  Exchange fed.topic:
    localhost:10004 <=> localhost:10003

Static Routes:
  none found
```

   Note that the two dynamic exchange links are displayed as though they were one bidirectional link. The `route map` command is helpful for larger, more complex networks.
2. Configure a network with 16 dynamic exchange routes:

```
qpid-route dynamic add localhost:10001 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10001 fed.topic

qpid-route dynamic add localhost:10003 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10003 fed.topic

qpid-route dynamic add localhost:10004 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10004 fed.topic

qpid-route dynamic add localhost:10002 localhost:10005 fed.topic
qpid-route dynamic add localhost:10005 localhost:10002 fed.topic

qpid-route dynamic add localhost:10005 localhost:10006 fed.topic
qpid-route dynamic add localhost:10006 localhost:10005 fed.topic

qpid-route dynamic add localhost:10006 localhost:10007 fed.topic
qpid-route dynamic add localhost:10007 localhost:10006 fed.topic

qpid-route dynamic add localhost:10006 localhost:10008 fed.topic
qpid-route dynamic add localhost:10008 localhost:10006 fed.topic
```

3. Use the `route map` command starting with any one broker to see the entire network:

```
$ qpid-route route map localhost:10001

Finding Linked Brokers:
    localhost:10001... Ok
    localhost:10002... Ok
    localhost:10003... Ok
    localhost:10004... Ok
    localhost:10005... Ok
    localhost:10006... Ok
    localhost:10007... Ok
    localhost:10008... Ok

Dynamic Routes:

  Exchange fed.topic:
    localhost:10002 <=> localhost:10001
    localhost:10003 <=> localhost:10002
    localhost:10004 <=> localhost:10002
    localhost:10005 <=> localhost:10002
    localhost:10006 <=> localhost:10005
    localhost:10007 <=> localhost:10006
    localhost:10008 <=> localhost:10006

Static Routes:
  none found
```

Report a bug

10.4.9. Resilient Connections

When a broker route is created, or when a durable broker route is restored after broker restart, a *resilient connections* is created between the source broker and the destination broker. If the connection fails due to a communication error, it attempts to reconnect. The retry interval begins at 2 seconds and, as more attempts are made, grows to 64 seconds. It continues to retry every 64 seconds thereafter. If the connection fails due to an authentication problem, it will not attempt to reconnect.

Report a bug

10.4.10. View Resilient Connections

The command `link list` can be used to show the resilient connections for a broker:

```
$ qpid-route link list localhost:10001

Host             Port    Transport Durable  State              Last Error
=============================================================================
localhost        10002   tcp           N     Operational
localhost        10003   tcp           N     Operational
localhost        10009   tcp           N     Waiting            Connection refused
```

In the above output, `Last Error` contains the string representation of the last connection error received for the connection. `State` represents the state of the connection, and may be one of the following values:

Table 10.2. State values in `$ qpid-route link list`

Option	Description
Waiting	Waiting before attempting to reconnect.
Connecting	Attempting to establish the connection.
Operational	The connection has been established and can be used.
Failed	The connection failed and will not retry (usually because authentication failed).
Closed	The connection has been closed and will soon be deleted.
Passive	If a cluster is federated to another cluster, only one of the nodes has an actual connection to remote node. Other nodes in the cluster have a passive connection.

Chapter 11. Qpid JCA Adapter

11.1. JCA Adapter

A JCA Adapter provides outbound and inbound connectivity between Enterprise Information Systems (for example, mainframe transaction processing and database systems), application servers, and enterprise applications. It controls the inflow of messages to Message-Driven Beans (MDBs) and the outflow of messages sent from other Java EE components. It also provides a variety of options to fine tune your messaging applications.

Report a bug

11.2. Qpid JCA Adapter

The Qpid Resource Adapter is a Java Connector Architecture (JCA) 1.5 compliant resource adapter that permits Java EE integration between EE applications and AMQP 0.10 message brokers. Currently the adapter only supports C++ based brokers and has only been tested with Apache Qpid C++ broker.

Report a bug

11.3. Install the Qpid JCA Adapter

For systems running Red Hat Enterprise Linux 5 or 6, the Qpid JCA Adapter is provided in the `qpid-jca` and `qpid-jca-xarecovery` packages. These RPM packages are included with the default MRG Messaging installation.

For other operating systems, the Qpid JCA Adapter is provided in the `JCA Adapter <JCA-VERSION>` and `JCA Adapter <JCA-VERSION> detached signature` packages. These ZIP files can be obtained from the `Downloads` section of the `MRG Messaging v. 2 (for non-Linux platforms)` channel on the Red Hat Network.

Report a bug

11.4. Qpid JCA Adapter Configuration

11.4.1. Per-Application Server Configuration Information

Deploying the components of a resource adapter varies with different application servers. This guide provides an overview of the adapter's capabilities, and installation instructions for JBoss Enterprise Application Platform 5 and 6. For most application server platforms, platform-specific details for configuring the adapter are provided in README files typically named `README-<server-platform>.txt`.

See Also:

- Section 11.5, “Deploying the Qpid JCA Adapter on JBoss EAP 5”
- Section 11.6, “Deploying the Qpid JCA Adapter on JBoss EAP 6”

Report a bug

11.4.2. Components of the JCA Adapter

As per the JCA specification, a JCA Resource Adapter contains four main components:

- The **ResourceAdapter JavaBean** provides a set of global configuration options.
- The **ManagedConnectionFactory JavaBean** is used to configure outbound connectivity options.
- The **ActivationSpec JavaBean** provides configuration options for inbound connectivity.
- **Administered Objects** are JavaBeans specific to the messaging provider.

When a ManagedConnectionFactory JavaBean or ActivationSpec JavaBean are deployed, they can inherit the configuration properties from the ResourceAdapter JavaBean. Alternatively, they can provide specific properties that will override the default properties.

Some properties from the ResourceAdapter, ManagedConnectionFactory and ActivationSpec JavaBeans are specific to the JCA Adapter. However, the majority of them directly correspond to the Qpid Java Message Service (JMS) client. Being familiar with the correct syntax and configuration options for the JMS client and the JCA 1.5 specification is recommended.

Report a bug

11.4.3. ResourceAdapter

11.4.3.1. ResourceAdapter JavaBean

The ResourceAdapter JavaBean provides global configuration options for inbound and outbound connectivity. Its properties can be found in the `META-INF/ra.xml` deployment descriptor provided with the resource adapter. This file is a J2EE/ JEE standard descriptor which contains the configuration template and defaults for ResourceAdapter, OutboundConnectors and InboundConnections.

Report a bug

11.4.3.2. ResourceAdapter JavaBean Properties

The following list describes the set of ResourceAdapter properties and their default configurations. These properties can be edited to suit your environment.

ResourceAdapter JavaBean Properties

`ClientId`

The unique client identifier. From the JMS API, this is used only in the context of durable subscriptions.

Default: client_id

`SetupAttempts`

The number of attempts the ResourceAdapter will make to successfully set up an inbound activation on deployment, or when an exception occurs at runtime.

Default: 5

`SetupInterval`

The interval, in milliseconds, the adapter waits between setup attempts.

Default: 5000

UseLocalTx
Whether or not to use local transactions instead of XA.

Default: false

UserName
The default user name.

Default: guest

Password
The default password.

Default: guest

Host
The broker hostname/ ip address.

Default: localhost

Port
The broker port.

Default: 5672

Path
The virtual path for the connection factory.

Default: test

ConnectionURL
The broker full connection URL.

Default: `amqp://anonymous:passwd@client/test?brokerlist='tcp://localhost?sasl_mechs='PLAIN''`

TransactionManagerLocatorClass
The class responsible for locating the transaction manager within a specific application server. This is a ResourceAdapter JavaBean specific property and is application server specific.

Default: none

TransactionManagerLocatorMethod
The specific method in the class above used to acquire a reference to the platform specific transaction manager. This is a ResourceAdapter JavaBean specific property and is application server specific.

Default: none

Report a bug

11.4.3.3. XA Support

If you require XA support, both the *TransactionManagerLocatorClass* and the *TransactionManagerLocatorMethod* properties must be set. While application servers typically provide a mechanism to do this in the form of a specific deployment descriptor or GUI console, the `META-INF/ra.xml` file can also be modified directly using any text editor.

Report a bug

11.4.4. ManagedConnectionFactory

11.4.4.1. ManagedConnectionFactory JavaBean

The ManagedConnectionFactory JavaBean provides outbound connectivity for the Qpid JCA Adapter. In addition to the properties inherited from the ResourceAdapter JavaBean, the ManagedConnectionFactory JavaBean provides specific properties only applicable to outbound connectivity.

Report a bug

11.4.4.2. ManagedConnectionFactory JavaBean Properties

The following list describes the set of ManagedConnectionFactory properties and their default configurations. These properties can be found in the `META-INF/ra.xml` file, and edited to suit your environment.

ManagedConnectionFactory JavaBean Properties

SessionDefaultType

The default type of Session. Currently unused.

Default: javax.jms.Queue

UseTryLock

Multi-purpose property used to specify that a lock on the underlying managed connection should be used, and the wait duration to acquire the lock. Primarily used for transaction management. A null or zero value will attempt to acquire the lock without a duration. A value greater than zero will wait *n* number of seconds before failing to acquire the lock.

Default: 0

KeyStorePassword

The KeyStore password for SSL.

Default: none

KeyStorePath

The path to the KeyStore.

Default: none

CertType

The type of certificate.

Default: SunX509

Report a bug

11.4.5. ActivationSpec

11.4.5.1. ActivationSpec JavaBean

The ActivationSpec JavaBean provides inbound connectivity for the Qpid JCA Adapter. In addition to the properties inherited from the ResourceAdapter JavaBean, the ActivationSpec JavaBean provides specific properties only applicable to inbound connectivity.

Report a bug

11.4.5.2. ActivationSpec JavaBean Properties

The following list describes the set of ActivationSpec properties and their default configurations. These properties can be found in the `META-INF/ra.xml` file, and edited to suit your environment.

ActivationSpec JavaBean Properties

`UseJNDI`

Whether or not to attempt looking up an inbound destination from Java Naming and Directory Interface (JNDI). If false, an attempt will be made to construct the destination from the other ActivationSpec properties.

Default: true

`Destination`

The name of the destination to listen for messages.

Default: none

`DestinationType`

The type of destination to listen on. Valid values are *`javax.jms.Queue`* or *`java.jms.Topic`*.

Default: none

`MessageSelector`

The JMS properties that will be used in selecting specific message. Please see the JMS specification for further details.

Default: none

`AcknowlegeMode`

Whether or not the client or consumer will acknowledge any messages it receives. Ignored in a transacted scenario. Please see the JMS specification for more details.

Default: AUTO_ACKNOWLEDGE

`SubscriptionDurablity`

Whether or not the subscription is durable.

Default: none

SubscriptionName
The name of the subscription.

Default: none

MaxSession
The maximum number of sessions the activation supports.

Default: 15

TransactionTimeout
The timeout for the XA transaction for inbound messages.

Default: 0

MaxPrefetch
The maximum number of message credits being pre-fetched. This property can be specified in two ways:

- As a connection URL parameter: *amqp://guest:guest@test/test?maxprefetch='100'&brokerlist='tcp://localhost:5672*
- Using the capacity property in Address Strings: *Ex my-queue; {create: always, link:{capacity: 10}}*

Default: 500

Report a bug

11.4.6. Administered Objects

11.4.6.1. Qpid JCA Adapter Administered Objects

The JCA specification provides for administered objects. As per the specification, administered objects are JavaBeans specific to the messaging provider. The Qpid JCA Adapter provides two administered objects — **QpidDestinationProxy**, used to configure JMS destinations; and **Transaction Support**, a specialized JMS Connection Factory.

Report a bug

11.4.6.2. QpidDestinationProxy

The QpidDestinationProxy allows a developer, deployer or administrator to create destinations (queues or topic) and bind these destinations into Java Naming and Directory Interface (JNDI).

Report a bug

11.4.6.3. QpidDestinationProxy Properties

destinationType

The type of destination to create. Valid values are *QUEUE* or *TOPIC*.

DestinationAddress

The address string of the destination. Please see the Qpid Java JMS client documentation for valid values.

QpidConnectionFactoryProxy

The QpidConnectionFactoryProxy allows a non-JCA ConnectionFactory to be bound into the JNDI tree. This ConnectionFactory can be used outside the application server. Typically a ConnectionFactory of this type is used by Swing or other two-tier clients which do not require JCA. The QpidConnectionFactoryProxy provides one property.

ConnectionURL

The URL used to configure the connection factory. Please see the Qpid Java Client documentation for further details.

Report a bug

11.4.6.4. Transaction Support

The Qpid JCA Adapter provides three levels of transaction support — **XA**, **LocalTransactions** and **NoTransaction**.

Typically, using the Qpid JCA Adapter implies the use of XA transactions, however this is not always the case. Transaction support configuration is specific to each application server, refer to the documentation for each supported application server.

Report a bug

11.4.6.5. Transaction Limitations

There are two limitations with the Qpid JCA Adapter at this time:

1. Currently, the Qpid C++ broker does not support the use of XA within the context of clustered brokers. If you are running a cluster, you must configure the adapter to use LocalTransactions.
2. XARecovery is currently not implemented. In case of a system failure, incomplete (or *in doubt*) transactions must be manually resolved by an administrator or other qualified personnel.

Report a bug

11.5. Deploying the Qpid JCA Adapter on JBoss EAP 5

11.5.1. Deploy the Qpid JCA adapter on JBoss EAP 5

To install the Qpid JCA Adapter for use with JBoss EAP 5, transfer its configuration files to the JBoss deploy directory.

Procedure 11.1. To deploy the Qpid JCA adapter for JBoss EAP

1. Locate the `qpid-ra-<version>.rar` file. It is a zip archive data file which contains the resource adapter, Qpid Java client `.jar` files and the `META-INF` directory.

2. Copy the qpid-ra-<*version*>.rar file to your JBoss deploy directory. The JBoss deploy directory is *JBOSS_ROOT*/server/<*server-name*>/deploy, where *JBOSS_ROOT* denotes the root directory of your JBoss installation and <*server-name*> denotes the name of your deployment server.
3. A successful adapter installation is accompanied by the following message:

```
INFO  [QpidResourceAdapter] Qpid resource adaptor started
```

At this point, the adapter is deployed and ready for configuration.

Report a bug

11.5.2. JCA Configuration on JBoss EAP 5

11.5.2.1. JCA Adapter Configuration File

The standard configuration mechanism for JCA 1.5 adapters is the ra.xml deployment descriptor. Like other EE-based descriptors, this file is located in the META-INF directory of the provided EE artifact (ie the .rar file). This file provides a set of configuration defaults.

The ResourceAdapter configuration provides generic properties for inbound and outbound connectivity. However, these properties can be overridden when deploying ManagedConnectionFactories and inbound activations using the standard JBoss configuration artifacts, the *-ds.xml file and MDB ActivationSpec. A sample *-ds.xml file, qpid-jca-ds.xml, is located in the directory, /usr/share/doc/qpid-jca-<*VERSION*>/example/conf/.

The Qpid JCA Adapter directory /usr/share/qpid-jca contains the general README.txt file, which provides a detailed description of all the properties associated with the Qpid JCA Adapter.

Report a bug

11.5.2.2. ConnectionFactory Configuration

11.5.2.2.1. ConnectionFactory

Compliant with the JCA specification, the ConnectionFactory component defines properties for standard outbound connectivity.

Report a bug

11.5.2.2.2. ConnectionFactory Configuration in EAP 5

In JBoss EAP 5, ConnectionFactories are configured using the *-ds.xml file. A sample file (qpid-jca-ds.xml) is provided with your distribution. This file can be modified to suit your development or deployment needs.

Report a bug

11.5.2.2.3. XAConnectionFactory Example

The following example describes the ConnectionFactory portion of the sample file for XA transactions:

```
<tx-connection-factory>
  <jndi-name>QpidJMSXA</jndi-name>
  <xa-transaction/>
  <rar-name>qpid-ra-<ra-version>.rar</rar-name>
  <connection-definition>org.apache.qpid.ra.QpidRAConnectionFactory</connection-
definition>
  <config-property name="ConnectionURL">amqp://guest:guest@/test?
brokerlist='tcp://localhost:5672?sasl_mechs='ANONYMOUS''</config-property>
  <max-pool-size>20</max-pool-size>
</tx-connection-factory>
```

- The *QpidJMSXA* connection factory defines an XA-capable ManagedConnectionFactory.
- You must insert your particular ra version for the *rar-name* property.
- The *jndi-name* and *ConnectionURL* properties can be modified to suit your environment.

After deployment, the ConnectionFactory will be bound into Java Naming and Directory Interface (JNDI) with the following syntax:

```
java:<jndi-name>
```

In this example, this would resolve to:

```
java:QpidJMSXA
```

Report a bug

11.5.2.2.4. Local ConnectionFactory Example

The following example describes the ConnectionFactory portion of the sample file for either local transactions:

```
<tx-connection-factory>
  <jndi-name>QpidJMS</jndi-name>
  <rar-name>qpid-ra-<ra-version>.rar</rar-name>
  <local-transaction/>
  <config-property name="useLocalTx" type="java.lang.Boolean">true</config-
property>
  <config-property name="ConnectionURL">amqp://anonymous:@client/test?
brokerlist='tcp://localhost:5672?sasl_mechs='ANONYMOUS''</config-property>
  <connection-definition>org.apache.qpid.ra.QpidRAConnectionFactory</connection-
definition>
  <max-pool-size>20<max-pool-size>
</tx-connection-factory>
```

- The *QpidJMS* connection factory defines a non-XA ConnectionFactory. Typically this is used as a specialized ConnectionFactory where XA is not desired, or if you are running with a clustered Qpid Broker configuration that currently does not support XA.
- You must insert your particular ra version for the *rar-name* property.
- The *jndi-name* and *ConnectionURL* properties can be modified to suit your environment.

After deployment, the ConnectionFactory will be bound into Java Naming and Directory Interface (JNDI) with the following syntax:

```
java:<jndi-name>
```

In this example, this would resolve to:

```
java:QpidJMS
```

Report a bug

11.5.2.3. Administered Object Configuration

11.5.2.3.1. Administered Objects in EAP 5

Destinations (queues, topics) are configured in JBoss EAP via JCA standard Administered Objects (AdminObjects). These objects are placed within the `*-ds.xml` file alongside your ConnectionFactory configurations. The sample `qpid-jca-ds.xml` file provides two such objects: JMS Queue/ Topic and Connection Factory.

Report a bug

11.5.2.3.2. JMS Queue Administered Object Example

```
<mbean code="org.jboss.resource.deployment.AdminObject"
       name="qpid.jca:name=HelloQueue">
   <attribute name="JNDIName">Hello</attribute>
   <depends optional-attribute-
name="RARName">jboss.jca:service=RARDeployment,name='qpid-ra-<ra-
version>.rar'</depends>
   <attribute name="Type">javax.jms.Destination</attribute>
   <attribute name="Properties">

destinationAddress=jms_ee_mdb_mdb_sndQ_MDB_QUEUE;{create:always,node:{type:queue}}
      destinationType=queue
   </attribute>
</mbean>
```

The above XML defines a JMS Queue which is bound into JNDI as:

```
queue/HelloQueue
```

This destination can be retrieved from JNDI and used for the consumption or production of messages. The *desinationAddress* property can be customized for your environment. Refer to Qpid Java Client documentation for specific configuration options.

Report a bug

11.5.2.3.3. JMS Topic Administered Object Example

```
<mbean code="org.jboss.resource.deployment.AdminObject"
       name="qpid.jca:name=HelloTopic">
   <attribute name="JNDIName">HelloTopic</attribute>
   <depends optional-attribute-
name="RARName">jboss.jca:service=RARDeployment,name='qpid-ra-<ra-
version>.rar'</depends>
   <attribute name="Type">javax.jms.Destination</attribute>
   <attribute name="Properties">

destinationAddress=jms_ee_mdb_mdb_sndQ_MDB_TOPIC;{create:always,node:{type:topic}}
      destinationType=topic
   </attribute>
</mbean>
```

The above XML defines a JMS Topic which is bound into JNDI as:

```
HelloTopic
```

This destination can be retrieved from JNDI and used for the consumption or production of messages. The *desinationAddress* property can be customized for your environment. Refer to Qpid Java Client documentation for specific configuration options.

Report a bug

11.5.2.3.4. ConnectionFactory Administered Object Example

```
<mbean code="org.jboss.resource.deployment.AdminObject"
       name="qpid.jca:name=QpidConnectionFactory">
   <attribute name="JNDIName">QpidConnectionFactory</attribute>
   <depends optional-attribute-
name="RARName">jboss.jca:service=RARDeployment,name='qpid-ra-<ra-
version>.rar'</depends>
   <attribute name="Type">javax.jms.ConnectionFactory</attribute>
   <attribute name="Properties">
      ConnectionURL=amqp://anonymous:@client/test?brokerlist='tcp://localhost:5672?
sasl_mechs='ANONYMOUS''
   </attribute>
</mbean>
```

The above XML defines a ConnectionFactory that can be used for JBoss EAP 5 and also other external clients. Typically, this connection factory is used by standalone or 'thin' clients which do not require an application server. This object is bound into the JBoss EAP 5 JNDI tree as:

```
QpidConnectionFactory
```

Report a bug

11.5.2.4. ActivationSpec Configuration

11.5.2.4.1. ActivationSpec Configuration

The standard method for inbound communication is the MessageDrivenBean architecture, configured via the ActivationSpec mechanism. Refer to the general `README.txt` file for an explanation of the QpidActivationSpec, as well as general inbound connectivity options.

An ActivationSpec can either be configured via the Java Annotation mechanism, or in the `ejb-jar.xml` deployment descriptor. An example application that shows both methods are provided with the JCA Adapter RPM, and can be found in `/usr/share/doc/qpid-java-jca/example`.

Report a bug

11.6. Deploying the Qpid JCA Adapter on JBoss EAP 6

11.6.1. Deploy the Qpid JCA Adapter on JBoss EAP 6

To install the Qpid JCA Adapter for use with JBoss EAP 6, transfer its configuration files to the JBoss deploy directory.

Procedure 11.2. To deploy the Qpid JCA adapter for JBoss EAP

1. Locate the `qpid-ra-<version>.rar` file. It is a zip archive data file that contains the

resource adapter, Qpid Java client `.jar` files and the `META-INF` directory.

2. Copy the `qpid-ra-<version>.rar` file to your JBoss deploy directory. The JBoss deploy directory is *JBOSS_ROOT*`/<server-config>/deployments`, where *JBOSS_ROOT* is the root directory of your JBoss installation and `<server-config>` is the name of your deployment server configuration.

When the JCA Adapter is deployed, it must be configured before it can be used.

Report a bug

11.6.2. JCA Configuration on JBoss EAP 6

11.6.2.1. JCA Adapter Configuration Files in JBoss EAP 6

JBoss EAP 6.x uses a different configuration scheme from previous EAP versions.

Each server configuration type contains the following configuration files in the directory *JBOSS_ROOT*`/<server-config>/configuration`:

- `<server-config>`-full.xml
- `<server-config>`-full-ha.xml
- `<server-config>`.xml

Each file corresponds to a set of capabilities, and contains the configuration of the subsystems for that server profile.

Report a bug

11.6.2.2. Replace the Default Messaging Provider with the Qpid JCA Adapter

The following XML fragment from a server configuration file replaces the default EAP 6 messaging provider:

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.2">
   <session-bean>
      <stateless>
         <bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>
      </stateless>
      <stateful default-access-timeout="5000" cache-ref="simple"/>
      <singleton default-access-timeout="5000"/>
   </session-bean>
   <mdb>
      <resource-adapter-ref resource-adapter-name="qpid-ra-<rar-version>.rar"/>
      <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
   </mdb>
   <pools>
      <bean-instance-pools>
         <strict-max-pool name="slsb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
         <strict-max-pool name="mdb-strict-max-pool" max-pool-size="20" instance-
acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      </bean-instance-pools>
   </pools>
   <caches>
      <cache name="simple" aliases="NoPassivationCache"/>
      <cache name="passivating" passivation-store-ref="file"
aliases="SimpleStatefulCache"/>
   </caches>
   <passivation-stores>
      <file-passivation-store name="file"/>
   </passivation-stores>
   <async thread-pool-name="default"/>
   <timer-service thread-pool-name="default">
      <data-store path="timer-service-data" relative-to="jboss.server.data.dir"/>
   </timer-service>
   <remote connector-ref="remoting-connector" thread-pool-name="default"/>
   <thread-pools>
      <thread-pool name="default">
         <max-threads count="10"/>
         <keepalive-time time="100" unit="milliseconds"/>
      </thread-pool>
   </thread-pools>
</subsystem>
```

The relevant lines in this sub-system configuration are:

```
<mdb>
   <resource-adapter-ref resource-adapter-name="qpid-ra-<rar-version>.rar"/>
   <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
</mdb>
```

Report a bug

11.6.2.3. Configuration Methods

You have two options:

1. Directly edit the existing configuration file.
2. Copy the existing configuration file, edit the copy, then start the server using the new configuration file with the command:

   ```
   JBOSS_HOME/bin/standalone.sh -c your-modified-config.xml
   ```

Report a bug

11.6.2.4. Example Minimal EAP 6 Configuration

The following XML fragment from a JBoss EAP 6 server configuration file is a minimal example configuration, configuring an XA aware ManagedConnectionFactory and two JMS destinations (queue and topic)

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
        <resource-adapter>
            <archive>
                qpid-ra-<rar-version>.rar
            </archive>
            <transaction-support>
                XATransaction
            </transaction-support>
            <config-property name="connectionURL">
                amqp://anonymous:passwd@client/test?brokerlist='tcp://localhost?
sasl_mechs='PLAIN''
            </config-property>
            <config-property name="TransactionManagerLocatorClass">
                org.apache.qpid.ra.tm.JBoss7TransactionManagerLocator
            </config-property>
            <config-property name="TransactionManagerLocatorMethod">
                getTm
            </config-property>
            <connection-definitions>
                <connection-definition class-
name="org.apache.qpid.ra.QpidRAManagedConnectionFactory" jndi-name="QpidJMSXA"
pool-name="QpidJMSXA">
                    <config-property name="connectionURL">
                        amqp://anonymous:passwd@client/test?
brokerlist='tcp://localhost?sasl_mechs='PLAIN''
                    </config-property>
                    <config-property name="SessionDefaultType">
                        javax.jms.Queue
                    </config-property>
                </connection-definition>
            </connection-definitions>
            <admin-objects>
                <admin-object class-name="org.apache.qpid.ra.admin.QpidTopicImpl"
jndi-name="java:jboss/exported/GoodByeTopic" use-java-context="false" pool-
name="GoodByeTopic">
                    <config-property name="DestinationAddress">
                        amq.topic/hello.Topic
                    </config-property>
                </admin-object>
                <admin-object class-name="org.apache.qpid.ra.admin.QpidQueueImpl"
jndi-name="java:jboss/exported/HelloQueue" use-java-context="false" pool-
name="HelloQueue">
                    <config-property name="DestinationAddress">
                        hello.Queue;{create:always, node:{type:queue, x-
declare:{auto-delete:true}}}
                    </config-property>
                </admin-object>
            </admin-objects>
        </resource-adapter>
    </resource-adapters>
</subsystem>
```

Report a bug

11.6.2.5. Further Resources

For further information, refer to the JBoss Enterprise Application Platform documentation and the README files included with the Qpid JCA Adapter.

Report a bug

Chapter 12. Management Tools and Consoles

12.1. Command-line utilities

12.1.1. Command-line Management utilities

The command-line tools are lightweight management and diagnostic tools designed for use at the shell prompt.

Table 12.1. Command-line Management utilities

Option	Description
qpid-config	Display and configure exchanges, queues, and bindings in the broker
qpid-tool	Access configuration, statistics, and control within the broker
qpid-queue-stats	Monitor the size and enqueue/dequeue rates of queues in a broker
qpid-cluster	Configure and view clusters
qpid-route	Configure federated routes among brokers
qpid-perftest	Measures throughput on a variety of scenarios, using adjustable parameters
qpid-stat	Display details and statistics for various broker objects
qpid-printevents	Subscribes to events from a broker and prints details of events raised to the console window
qpid-cluster-store	Used in recovering persistent data after a non-clean cluster shutdown

Report a bug

12.1.2. Changes in qpid-config and qpid-stat for MRG 2.3

In MRG 2.3 the `qpid-config` tool has been updated to align it with the latest developments in open source development. The following changes have been made:

Changes to `qpid-config` in MRG 2.3

- The `-a --broker-addr` command line option was changed to `-b --broker`. This option specifies the broker address.
- The `-b --bindings` command line option was changed to `-r --recursive`. This option shows the queue/exchange bindings.
- The `-a` command for the broker address, as well as the `--broker-addr` and `--bindings` form of the commands are still supported but are intentionally not in the usage help

Changes to `qpid-stat` in MRG 2.3

- Command line options are now presented in three groups: general options, command options and display options.
- The new way to specify the broker address is the `-b --broker` option. It is now under the `-g` options

Report a bug

12.1.3. Using qpid-config

Changes

1. View the full list of commands by running the `qpid-config --help` command from the shell prompt:

```
$ qpid-config --help

Usage:  qpid-config [OPTIONS]
qpid-config [OPTIONS] exchanges [filter-string]
qpid-config [OPTIONS] queues    [filter-string]
qpid-config [OPTIONS] add exchange <type> <name> [AddExchangeOptions]
qpid-config [OPTIONS] del exchange <name>
..[output truncated]...
```

2. View a summary of all exchanges and queues by using the `qpid-config` without options:

```
$ qpid-config

Total Exchanges: 6
   topic: 2
 headers: 1
  fanout: 1
  direct: 2
   Total Queues: 7
 durable: 0
    non-durable: 7
```

3. List information on all existing queues by using the queues command:

```
$ qpid-config queues
Queue Name                                    Attributes
================================================================
my-queue                                       --durable
qmfc-v2-hb-localhost.localdomain.20293.1  auto-del excl --limit-policy=ring
qmfc-v2-localhost.localdomain.20293.1     auto-del excl
qmfc-v2-ui-localhost.localdomain.20293.1  auto-del excl --limit-policy=ring
reply-localhost.localdomain.20293.1       auto-del excl
topic-localhost.localdomain.20293.1       auto-del excl --limit-policy=ring
```

4. Add new queues with the `add queue` command and the name of the queue to create:

```
$ qpid-config add queue queue_name
```

5. To delete a queue, use the `del queue` command with the name of the queue to remove:

```
$ qpid-config del queue queue_name
```

6. **Version 2.2 and below**
   List information on all existing exchanges with the exchanges command. Add the `-b` option to also see binding information:

```
$ qpid-config -b exchanges

Exchange '' (direct)
 bind pub_start => pub_start
 bind pub_done => pub_done
 bind sub_ready => sub_ready
 bind sub_done => sub_done
 bind perftest0 => perftest0
 bind mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => mgmt-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
 bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
Exchange 'amq.direct' (direct)
 bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
 bind repl-df06c7a6-4ce7-426a-9f66-da91a2a6a837 => repl-df06c7a6-4ce7-
426a-9f66-da91a2a6a837
 bind repl-c55915c2-2fda-43ee-9410-b1c1cbb3e4ae => repl-c55915c2-2fda-
43ee-9410-b1c1cbb3e4ae
Exchange 'amq.topic' (topic)
Exchange 'amq.fanout' (fanout)
Exchange 'amq.match' (headers)
Exchange 'qpid.management' (topic)
 bind mgmt.# => mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15
```

Version 2.3 and above
List information on all existing exchanges with the `exchanges` command. Add the `-r` option ("recursive") to also see binding information:

```
$ qpid-config -r exchanges

Exchange '' (direct)
 bind pub_start => pub_start
 bind pub_done => pub_done
 bind sub_ready => sub_ready
 bind sub_done => sub_done
 bind perftest0 => perftest0
 bind mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => mgmt-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
 bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
Exchange 'amq.direct' (direct)
 bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-
4a30-82ea-e76f50dd7d15
 bind repl-df06c7a6-4ce7-426a-9f66-da91a2a6a837 => repl-df06c7a6-4ce7-
426a-9f66-da91a2a6a837
 bind repl-c55915c2-2fda-43ee-9410-b1c1cbb3e4ae => repl-c55915c2-2fda-
43ee-9410-b1c1cbb3e4ae
Exchange 'amq.topic' (topic)
Exchange 'amq.fanout' (fanout)
Exchange 'amq.match' (headers)
Exchange 'qpid.management' (topic)
 bind mgmt.# => mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15
```

7. Add new exchanges with the `add exchange` command. Specify the type (`direct`, `topic` or `fanout`) along with the name of the exchange to create. You can also add the `--durable` option to make the exchange durable:

```
$ qpid-config add exchange direct exchange_name --durable
```

8. To delete an exchange, use the `del exchange` command with the name of the exchange to remove:

```
$ qpid-config del exchange exchange_name
```

Report a bug

12.1.4. Using qpid-cluster

`qpid-cluster` is a command-line utility that allows you to view information on a cluster and its brokers, disconnect a client connection, shut down a broker in a cluster, or shut down the entire cluster. You can see the options using the `--help` option:

```
$ qpid-cluster --help
```

```
Usage:  qpid-cluster [OPTIONS] [broker-addr]

             broker-addr is in the form:   [username/password@] hostname | ip-
address [:<port>]
             ex:  localhost, 10.1.1.7:10000, broker-host:10000,
guest/guest@localhost

Options:
          -C [--all-connections]  View client connections to all cluster members
          -c [--connections] ID   View client connections to specified member
          -d [--del-connection] HOST:PORT
                                   Disconnect a client connection
          -s [--stop] ID          Stop one member of the cluster by its ID
          -k [--all-stop]         Shut down the whole cluster
          -f [--force]             Suppress the 'are-you-sure?' prompt
          -n [--numeric]          Don't resolve names
```

Connect to a cluster and display basic information about the cluster and its brokers. When you connect to the cluster using `qpid-cluster`, you can use the host and port for any broker in the cluster. For instance, if a broker in the cluster is running on `localhost` on port 6664, you can start `qpid-cluster` like this:

```
$ qpid-cluster localhost:6664
```

Here is the output:

```
  Cluster Name: local_test_cluster
Cluster Status: ACTIVE
  Cluster Size: 3
       Members: ID=127.0.0.1:13143
URL=amqp:tcp:192.168.1.101:6664,tcp:192.168.122.1:6664,tcp:10.16.10.62:6664
               : ID=127.0.0.1:13167
URL=amqp:tcp:192.168.1.101:6665,tcp:192.168.122.1:6665,tcp:10.16.10.62:6665
               : ID=127.0.0.1:13192
URL=amqp:tcp:192.168.1.101:6666,tcp:192.168.122.1:6666,tcp:10.16.10.62:6666
```

The ID for each broker in cluster is given on the left. For instance, the ID for the first broker in the cluster is `127.0.0.1:13143`. The URL in the output is the broker's advertised address. Use the ID to shut down the broker using the `--stop` command:

```
$ qpid-cluster localhost:6664 --stop 127.0.0.1:13143
```

Report a bug

12.1.5. Using qpid-tool

1. The `qpid-tool` creates a connection to a broker, and commands are run within the tool, rather than at the shell prompt itself. To create the connection, run the `qpid-tool` at the shell prompt with the name or IP address of the machine running the broker you wish to view. You can also append a TCP port number with a `:` character:

```
$ qpid-tool localhost

Management Tool for QPID
qpid:
```

2. If the connection is successful, qpid-tool will display a `qpid:` prompt. Type `help` at this prompt to see the full list of commands:

```
qpid: help
Management Tool for QPID

Commands:
list                    - Print summary of existing objects by class
list <className>        - Print list of objects of the specified class
list <className> all    - Print contents of all objects of specified class
...[output truncated}...
```

3. `qpid-tool` uses the word *objects* to refer to queues, exchanges, brokers and other such devices. To view a list of all existing objects, type `list` at the prompt:

```
# qpid-tool
Management Tool for QPID
qpid: list
Summary of Objects by Type:
    Package                  Class          Active  Deleted
    ========================================================
    org.apache.qpid.broker  exchange       8       0
    com.redhat.rhm.store    store          1       0
    org.apache.qpid.broker  broker         1       0
    org.apache.qpid.broker  binding        16      12
    org.apache.qpid.broker  session        2       1
    org.apache.qpid.broker  connection     2       1
    org.apache.qpid.broker  vhost          1       0
    org.apache.qpid.broker  queue          6       5
    org.apache.qpid.broker  system         1       0
    org.apache.qpid.broker  subscription   6       5
```

4. You can choose which objects to list by also specifying a class:

```
qpid: list system
Object Summary:
    ID   Created   Destroyed   Index

=================================================================================
    167  07:34:13  -           UUID('b3e2610e-5420-49ca-8306-dca812db647f')
```

5. To view details of an object class, use the `schema` command and specify the class:

```
qpid: schema queue
Schema for class 'qpid.queue':
Element                 Type          Unit         Access      Notes
Description
================================================================================
=========================================
vhostRef                reference                  ReadCreate  index
name                    short-string               ReadCreate  index
durable                 boolean                    ReadCreate
autoDelete              boolean                    ReadCreate
exclusive               boolean                    ReadCreate
arguments               field-table                ReadOnly
Arguments supplied in queue.declare
storeRef                reference                  ReadOnly
Reference to persistent queue (if durable)
msgTotalEnqueues        uint64        message                        Total
messages enqueued
msgTotalDequeues        uint64        message                        Total
messages dequeued
msgTxnEnqueues          uint64        message
Transactional messages enqueued
msgTxnDequeues          uint64        message
Transactional messages dequeued
msgPersistEnqueues      uint64        message
Persistent messages enqueued
msgPersistDequeues      uint64        message
Persistent messages dequeued
msgDepth                uint32        message
Current size of queue in messages
msgDepthHigh            uint32        message
Current size of queue in messages (High)
msgDepthLow             uint32        message
Current size of queue in messages (Low)
byteTotalEnqueues       uint64        octet                          Total
messages enqueued
byteTotalDequeues       uint64        octet                          Total
messages dequeued
byteTxnEnqueues         uint64        octet
Transactional messages enqueued
byteTxnDequeues         uint64        octet
Transactional messages dequeued
bytePersistEnqueues     uint64        octet
Persistent messages enqueued
bytePersistDequeues     uint64        octet
Persistent messages dequeued
byteDepth               uint32        octet
Current size of queue in bytes
byteDepthHigh           uint32        octet
Current size of queue in bytes (High)
byteDepthLow            uint32        octet
Current size of queue in bytes (Low)
enqueueTxnStarts        uint64        transaction                    Total
enqueue transactions started
enqueueTxnCommits       uint64        transaction                    Total
enqueue transactions committed
enqueueTxnRejects       uint64        transaction                    Total
enqueue transactions rejected
enqueueTxnCount         uint32        transaction                    Current
pending enqueue transactions
enqueueTxnCountHigh     uint32        transaction                    Current
pending enqueue transactions (High)
enqueueTxnCountLow      uint32        transaction                    Current
pending enqueue transactions (Low)
dequeueTxnStarts        uint64        transaction                    Total
```

```
dequeue transactions started
dequeueTxnCommits      uint64        transaction                    Total
dequeue transactions committed
dequeueTxnRejects      uint64        transaction                    Total
dequeue transactions rejected
dequeueTxnCount        uint32        transaction                    Current
pending dequeue transactions
dequeueTxnCountHigh    uint32        transaction                    Current
pending dequeue transactions (High)
dequeueTxnCountLow     uint32        transaction                    Current
pending dequeue transactions (Low)
consumers              uint32        consumer
Current consumers on queue
consumersHigh          uint32        consumer
Current consumers on queue (High)
consumersLow           uint32        consumer
Current consumers on queue (Low)
bindings               uint32        binding
Current bindings
bindingsHigh           uint32        binding
Current bindings (High)
bindingsLow            uint32        binding
Current bindings (Low)
unackedMessages        uint32        message
Messages consumed but not yet acked
unackedMessagesHigh    uint32        message
Messages consumed but not yet acked (High)
unackedMessagesLow     uint32        message
Messages consumed but not yet acked (Low)
messageLatencySamples  delta-time    nanosecond                     Broker
latency through this queue (Samples)
messageLatencyMin      delta-time    nanosecond                     Broker
latency through this queue (Min)
messageLatencyMax      delta-time    nanosecond                     Broker
latency through this queue (Max)
messageLatencyAverage  delta-time    nanosecond                     Broker
latency through this queue (Average)
```

6. To exit the tool and return to the shell, type `quit` at the prompt:

```
qpid: quit
Exiting...
```

Report a bug

12.1.6. Using qpid-queue-stats

Run the command `qpid-queue-stats` to launch the tool.

The tool will begin to report stats for the broker on the current machine, with the following format:

```
Queue Name                                       Sec       Depth     Enq Rate
Deq Rate
================================================================================
=========
message_queue                                   10.00      11224         0.00
54.01
qmfc-v2-ui-radhe.26001.1                        10.00          0         0.10
0.10
topic-radhe.26001.1                             10.00          0         0.20
0.20
message_queue                                   10.01       9430         0.00
179.29
qmfc-v2-ui-radhe.26001.1                        10.01          0         0.10
0.10
topic-radhe.26001.1                             10.01          0         0.20
0.20
```

The queues on the broker are listed on the left. The *Sec* column is the sample rate. The tool is hardcoded to poll the broker in 10 second intervals. The *Depth* column reports the number of messages in the queue. *Enq Rate* is the number of messages added to the queue (enqueued) since the last sample. *Deq Rate* is the number of messages removed from the queue (dequeued) since the last sample.

To view the queues on another server, use the `-a` switch, and provide a remote server address, and optionally the remote port and authentication credentials.

For example, to examine the queues on a remote server with the address 192.168.1.145, issue the command:

```
qpid-queue-stats -a 192.168.1.145
```

To examine the queues on the server broker1.mydomain.com:

```
qpid-queue-stats -a broker1.mydomain.com
```

To examine the queues on the server broker1.mydomain.com, where the broker is running on port 8888:

```
qpid-queue-stats -a broker1.mydomain.com:8888
```

To examine the queues on the server 192.168.1.145, which requires authentication:

```
qpid-queue-stats -a username/password@192.168.1.145
```

Report a bug

Exchange and Queue Declaration Arguments

A.1. Exchange and Queue Argument Reference

Changes

Following is a complete list of arguments for declaring queues and exchanges.

Exchange options

`qpid.exclusive-binding` **(bool)**

Ensures that a given binding key is associated with only one queue.

`qpid.ive` **(bool)**

If set to “true”, the exchange is an *initial value exchange*, which differs from other exchanges in only one way: the last message sent to the exchange is cached, and if a new queue is bound to the exchange, it attempts to route this message to the queue, if the message matches the binding criteria. This allows a new queue to use the last received message as an initial value.

`qpid.msg_sequence` **(bool)**

If set to “true”, the exchange inserts a sequence number named “qpid.msg_sequence” into the message headers of each message. The type of this sequence number is int64. The sequence number for the first message routed from the exchange is 1, it is incremented sequentially for each subsequent message. The sequence number is reset to 1 when the qpid broker is restarted.

`qpid.sequence_counter` **(int64)**

Start `qpid.msg_sequence` counting at the given number.

Queue options

`no-local` **(bool)**

Specifies that the queue should discard any messages enqueued by sessions on the same connection as that which declares the queue.

`qpid.alert_count` **(uint32_t)**

If the queue message count goes above this size an alert should be sent.

`qpid.alert_repeat_gap` **(int64_t)**

Controls the minimum interval between events in seconds. The default value is 60 seconds.

`qpid.alert_size` **(int64_t)**

If the queue size in bytes goes above this size an alert should be sent.

`qpid.auto_delete_timeout` **(bool)**

If a queue is configured to be automatically deleted, it will be deleted after the amount of

seconds specified here.

`qpid.browse-only` **(bool)**
All users of queue are forced to browse. Limit queue size with ring, LVQ, or TTL. Note that this argument name uses a hyphen rather than an underscore.

`qpid.file_count` **(int)**
Set the number of files in the persistence journal for the queue. Default value is 8.

`qpid.file_size` **(int64)**
Set the number of pages in the file (each page is 64KB). Default value is 24.

`qpid.flow_resume_count` **(uint32_t)**
Flow resume threshold value as a message count.

`qpid.flow_resume_size` **(uint64_t)**
Flow resume threshold value in bytes.

`qpid.flow_stop_count` **(uint32_t)**
Flow stop threshold value as a message count.

`qpid.flow_stop_size` **(uint64_t)**
Flow stop threshold value in bytes.

`qpid.last_value_queue` **(bool)**
Enables last value queue behavior.

`qpid.last_value_queue_key` **(string)**
Defines the key to use for a last value queue.

`qpid.last_value_queue_no_browse` **(bool)**
Enables special mode for last value queue behavior.

`qpid.max_count` **(uint32_t)**
The maximum byte size of message data that a queue can contain before the action dictated by the `policy_type` is taken.

`qpid.max_size` **(uint64_t)**
The maximum number of messages that a queue can contain before the action dictated by the `policy_type` is taken.

`qpid.msg_sequence` **(bool)**
Causes a sequence number to be added to headers of enqueued messages.

`qpid.optimistic_consume` **(bool)**
Allows the consumer to dequeue the message before the broker has acknowledged the producer, in order to reduce latency for durable messaging.

`qpid.persist_last_node` **(bool)**

Allows for a queue to treat all transient messages as persistent when a cluster fails down to a single node. When additional nodes in the cluster are restored, the transient messages will no longer be persisted. This mode will not be triggered if a cluster is started with only one active node, and the queues in this mode must be configured to be durable.

`qpid.policy_type` **(string)**
Sets default behavior for controlling queue size. Valid values are *`reject`*, *`flow_to_disk`*, *`ring`*, and *`ring_strict`*.

`qpid.priorities` **(size_t)**
The number of distinct priority levels recognized by the queue (up to a maximum of 10). The default value is 1 level.

`qpid.queue_event_generation` **(type: int)**
If the queue is created within a program, sets the queue options to enable queue events. Use the value `1` to replicate only enqueue events, or `2` to replicate both enqueue and dequeue events.

`qpid.trace.exclude` **(string)**
Does not send on messages which include one of the given (comma separated) trace ids.

`qpid.trace.id` **(string)**
Adds the given trace id as to the application header "`x-qpid.trace`" in messages sent from the queue.

`x-qpid-maximum-message-count`
This is an alias for `qpid.alert_count`.

`x-qpid-maximum-message-size`
This is an alias for `qpid.alert_size`.

`x-qpid-minimum-alert-repeat-gap`
This is an alias for `qpid.alert_repeat_gap`.

`x-qpid-priorities`
This is an alias for `qpid.priorities`.

Report a bug

OpenSSL Certificate Reference

B.1. Reference of Certificates

This reference for creating and managing certificates with the `openssl` command assumes familiarity with SSL. For more background information on SSL refer to the OpenSSL documentation at www.openssl.org.

Important

It is recommended that only certificates signed by an authentic Certificate Authority (CA) are used for secure systems. Instructions in this section for generating self-signed certificates are meant to facilitate test and development activities or evaluation of software while waiting for a certificate from an authentic CA.

Generating Certificates

Procedure B.1. Create a Private Key

- Use this command to generate a 1024-bit RSA private key with file encryption. If the key file is encrypted, the password will be needed every time an application accesses the private key.

```
# openssl genrsa -des3 -out mykey.pem 1024
```

Use this command to generate a key without file encryption:

```
# openssl genrsa  -out mykey.pem 1024
```

Procedure B.2. Create a Self-Signed Certificate

Each of the following commands generates a new private key and a *self-signed* certificate, which acts as its own CA and does not need additional signatures. This certificate expires one week from the time it is generated.

1. The *nodes* option causes the key to be stored without encryption. OpenSSL will prompt for values needed to create the certificate.

```
# openssl req -x509 -nodes -days 7 -newkey rsa:1024 -keyout mykey.pem -out
mycert.pem
```

2. The *subj* option can be used to specify values and avoid interactive prompts, for example:

```
# openssl req -x509 -nodes -days 7 -subj
'/C=US/ST=NC/L=Raleigh/CN=www.redhat.com' -newkey rsa:1024 -keyout mykey.pem
-out mycert.pem
```

3. The *new* and *key* options generate a certificate using an existing key instead of generating a new one.

```
# openssl req -x509 -nodes -days 7 -new -key mykey.pem -out mycert.pem
```

Create a Certificate Signing Request

To generate a certificate and have it signed by a Certificate Authority (CA), you need to generate a certificate signing request (CSR):

```
# openssl req -new -key mykey.pem -out myreq.pem
```

The certificate signing request can now be sent to an authentic Certificate Authority for signing and a valid signed certificate will be returned. The exact procedure to send the CSR and receive the signed certificate depend on the particular Certificate Authority you use.

Create Your Own Certificate Authority

You can create your own Certificate Authority and use it to sign certificate requests. If the Certificate Authority is added as a trusted authority on a system, any certificates signed by the Certificate Authority will be valid on that system. This option is useful if a large number of certificates are needed temporarily.

1. Create a self-signed certificate for the CA, as described in Procedure B.2, “Create a Self-Signed Certificate”.
2. OpenSSL needs the following files set up for the CA to sign certificates. On a Red Hat Enterprise Linux system with a fresh OpenSSL installation using a default configuration, set up the following files:
   a. Set the path for the CA certificate file as `/etc/pki/CA/cacert.pem`.
   b. Set the path for the CA private key file as `/etc/pki/CA/private/cakey.pem`.
   c. Create a zero-length index file at `/etc/pki/CA/index.txt`.
   d. Create a file containing an initial serial number (for example, 01) at `/etc/pki/CA/serial`.
   e. The following steps must be performed on RHEL 5:
      a. Create the directory where new certificates will be stored: `/etc/pki/CA/newcerts`.
      b. Change to the certificate directory: `cd /etc/pki/tls/certs`.
3. The following command signs a CSR using the CA:

   ```
   # openssl ca -notext -out mynewcert.pem -infiles myreq.pem
   ```

Install a Certificate

1. For OpenSSL to recognize a certificate, a hash-based symbolic link must be generated in the `certs` directory. `/etc/pki/tls` is the parent of the `certs` directory in Red Hat Enterprise Linux's version of OpenSSL. Use the `version` command to check the parent directory:

   ```
   # openssl version -d
   OPENSSLDIR: "/etc/pki/tls"
   ```

2. Create the required symbolic link for a certificate using the following command:

   ```
   # ln -s certfile `openssl x509 -noout -hash -in certfile`.0
   ```

   It is possible for more than one certificate to have the same hash value. If this is the case, change the suffix on the link name to a higher number. For example:

   ```
   # ln -s certfile `openssl x509 -noout -hash -in certfile`.4
   ```

Examine Values in a Certificate

The content of a certificate can be seen in plain text with this command:

```
# openssl x509 -text -in mycert.pem
```

Exporting a Certificate from NSS into PEM Format

Certificates stored in an NSS certificate database can be exported and converted to PEM format in several ways:

- This command exports a certificate with a specified nickname from an NSS database:

  ```
  # certutil -d . -L -n "Some Cert" -a > somecert.pem
  ```

- These commands can be used together to export certificates and private keys from an NSS database and convert them to PEM format. They produce a file containing the client certificate, the certificate of its CA, and the private key.

  ```
  # pk12util -d . -n "Some Cert" -o somecert.pk12
  # openssl pkcs12 -in somecert.pk12 -out tmckay.pem
  ```

  See documentation for the `openssl pkcs12` command for options that limit the content of the PEM output file.

Report a bug

Revision History

Revision 1.0.0-55 Tue Feb 19 2013 Joshua Wulf

Built from Content Specification: 7069, Revision: 375616 by jwulf