Red Hat Enterprise MRG 2.0

Messaging User Guide

Use, configuration and tuning information for MRG Messaging

Lana Brindley

Alison Young

Red Hat Enterprise MRG 2.0 Messaging User Guide
Use, configuration and tuning information for MRG Messaging Edition 1

Author	Lana Brindley	lbrindle@redhat.com
Author	Alison Young	alyoung@redhat.com

Copyright © 2011 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at *http://creativecommons.org/licenses/by-sa/3.0/*. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

All other trademarks are the property of their respective owners.

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

This book discusses the use, tuning and configuration of the MRG Messaging component of the Red Hat Enterprise MRG distributed computing platform. For installation instructions, see the *Messaging Installation Guide*. To learn how to program MRG Messaging applications, see the guide entitled *Programming in Apache Qpid*.

Preface

Red Hat Enterprise MRG

This book contains information on the use, tuning and configuration for the MRG Messaging component of Red Hat Enterprise MRG. Red Hat Enterprise MRG is a high performance distributed computing platform consisting of three components:

1. *M*essaging — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *R*ealtime — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *G*rid — Distributed High Throughput (HTC) and High Performance Computing (HPC).

All three components of Red Hat Enterprise MRG are designed to be used as part of the platform, but can also be used separately.

MRG Messaging

MRG Messaging is an open source, high performance, reliable messaging distribution that implements the Advanced Message Queuing Protocol (AMQP) standard. MRG Messaging is based on *Apache Qpid*[1].

This guide shows you how to use, configure and tune MRG Messaging. For installation instructions, see the *Messaging Installation Guide*. If you want to write your own applications for use with MRG Messaging, look at the guide entitled *Programming in Apache Qpid*.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[2] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

[1] http://qpid.apache.org
[2] https://fedorahosted.org/liberation-fonts/

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F2** to switch to the first virtual terminal. Press **Ctrl**+**Alt**+**F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

> File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/* against the product **Red Hat Enterprise MRG.**

When submitting a bug report, be sure to mention the manual's identifier: *Messaging_User_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction to MRG Messaging

In most environments, programs communicate by exchanging messages, much as people communicate by exchanging email. Unlike email, enterprise messaging systems provide guaranteed delivery, speed, security, and freedom from spam. Until recently, there was no protocol standard for enterprise messaging systems, so programmers either wrote their own, limited themselves to one language or platform, or used proprietary systems. Java JMS provided Java programmers with a common API, but Java JMS does not specify a protocol, so implementations are not interoperable unless they agree on a protocol, and Java JMS does not provide even an API for other languages.

AMQP (Advanced Message Queuing Protocol) is an open protocol standard for enterprise messaging, designed to make mission critical messaging widely available as a standard service, and to make enterprise messaging interoperable across platforms, programming languages, and vendors. Information about AMQP is available at the *AMQP website*[1]. The AMQP 0-10 specification is *available for download*[2].

MRG Messaging is a high-speed reliable messaging distribution for Linux that includes an AMQP 0-10 messaging broker, AMQP 0-10 client libraries for C++, Java JMS, and Python, persistence libraries, and management tools. MRG Messaging is part of Red Hat Enterprise MRG, which provides Messaging, Realtime, and Grid functionality. Each of these components can also be used separately.

Apache Qpid (*http://qpid.apache.org*[3]) is an Apache project that implements the AMQP protocol. MRG Messaging includes messaging components from Qpid.

1.1. AMQP Features

AMQP is the first open protocol standard for Enterprise Messaging. It is designed to support messaging for just about any distributed or business application. Routing can be configured flexibly, easily supporting common messaging paradigms like point-to-point, fanout, publish-subscribe, and request-response. AMQP messages can contain any kind of payload, including text, structured binary data, streaming media, or XML.

AMQP is carefully designed to avoid introducing performance bottlenecks or scalability issues, permitting highly optimized implementations that scale well as the number of clients, messages, or bytes increase. MRG Messaging can achieve hundreds of thousands of messages per second on typical servers, and over a million on highly optimized high-end servers communicating via Infiniband.

AMQP supports guaranteed delivery, transactions, distributed transactions, SASL client authentication, and replay and recovery in the event of a network failure.

1.2. The AMQP 0-10 Model

AMQP defines both a wire level protocol (the transport layer) and higher level semantics for messaging (the functional layer).

In AMQP, a *connection* represents a network connection, and a *session* represents the interface between a client and a broker. A session uses a connection for communication. Sessions may be synchronous or asynchronous.

[1] http://www.amqp.org/
[2] http://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1
[3] http://qpid.apache.org/

The following diagram shows how the MRG Messaging broker is used by producer-consumer applications. A *broker* contains exchanges and queues. Message producers write to exchanges, exchanges route messages to queues, and message consumers read from queues.

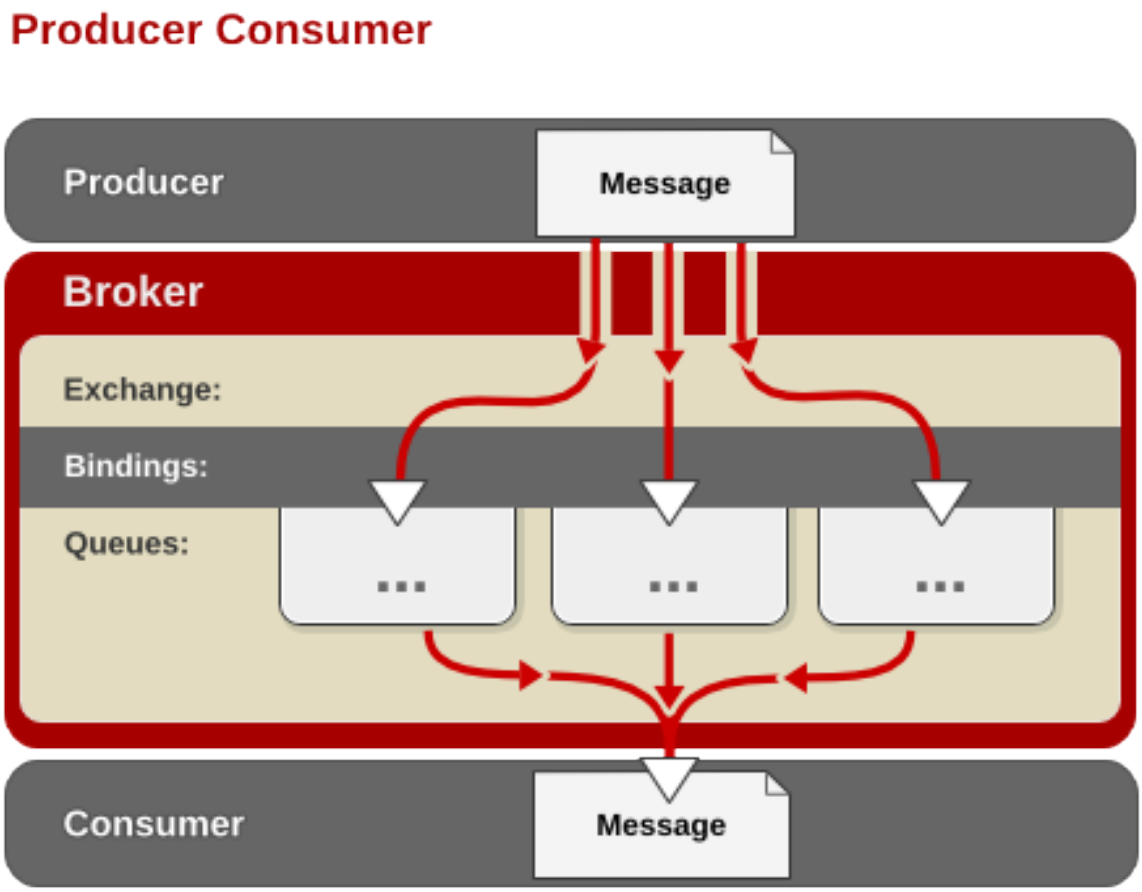


Figure 1.1. The AMQP 0-10 Model

A *message producer* creates a message, fills it with content, gives the message a routing key, and sends it to an exchange (for one kind of exchange, the fanout exchange, a routing key is optional). The *routing key* is simply a string that the exchange can use to determine to which queues the message should be delivered. The way the routing key is used depends on the exchange type and on the binding, which is discussed in the following paragraph. Before delivering a message, the message producer can also set various *message properties* in the message; for instance, one property determines whether the message is durable. A MRG Messaging broker does not lose durable messages even if the broker suffers a hardware failure. All durable messages are delivered when the broker is restarted. Another property can be used to specify message priority; the broker gives higher priority messages precedence.

An *exchange* accepts messages from message producers and routes them to message queues if the message meets the criteria expressed in a *binding*. A binding defines the relationship between an exchange and a message queue, specifying which messages should be routed to a given queue. If a queue is not bound to an exchange, it does not receive any messages from that exchange. The criteria found in a binding depend on the exchange type; for instance, in a direct exchange a binding might state that all messages with a given routing key should be sent to a particular queue, in a topic exchange a binding might state that a message whose routing key matches certain topics should be sent to a particular queue. This is discussed in more detail in *Chapter 2, Exchanges*.

A *message queue* holds messages and delivers them to the *message consumers* that subscribe to the queue. A message consumer can create, subscribe to, share, use, or destroy message queues (as long as they have permission to do so). A message queue may be *durable*, which means that the queue is never lost; even if the MRG Messaging Broker were to suffer a hardware failure, the queue would be restored when the broker is restarted. A message queue may be *exclusive*, which means only one client can consume messages from it. A message queue may also be *auto-delete*, which means that the queue will disappear from the server when the last client unsubscribes from the queue.

A message producer can use *transactions* to ensure that a group of messages are all received. In a transaction, messages and acknowledgments are batched together, and all messages in the transaction succeed or fail as a unit.

Exchanges

In the AMQP 0-10 model, a message producer sends messages to an exchange, which immediately distributes messages to queues. The exchange decides which queues should receive a message based on exchange type and the queue bindings, which bind a queue to an exchange. The AMQP 0-10 model is discussed in *Section 1.2, “The AMQP 0-10 Model”*.

In MRG Messaging there are four standard exchange types and a custom exchange type:

AMQP 0-10 Exchange Types

- Fanout
- Direct
- Default
- Topic
- Headers
- XML Exchange (A Custom Exchange)

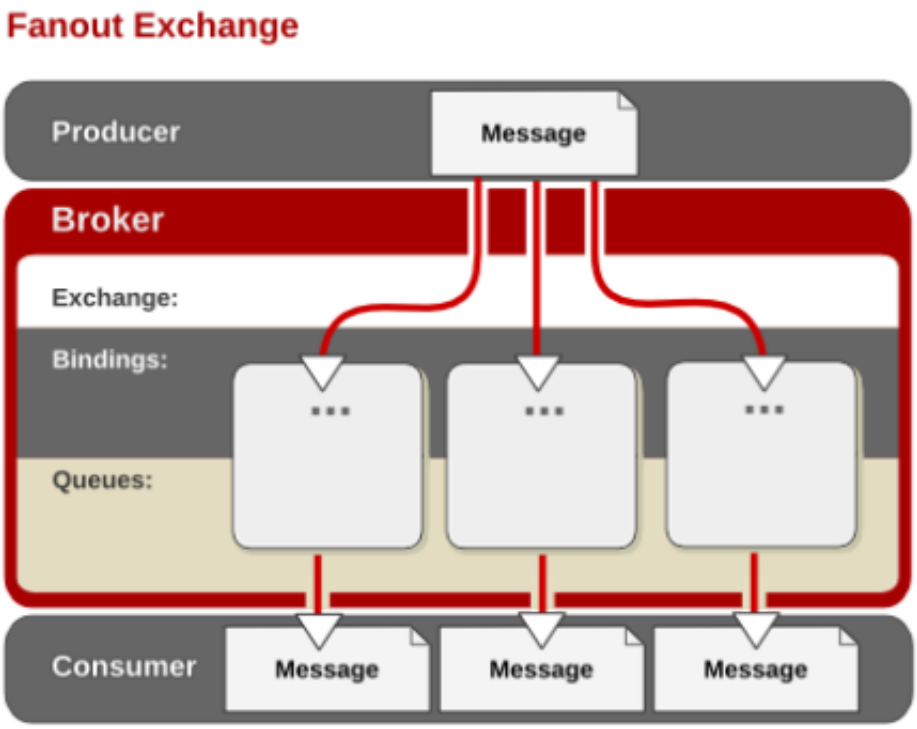
The semantics of these exchange types are described in the following sections.

2.1. Exchange Types

In the AMQP 0-10 model, exchange types differ only in the algorithm used to determine which queues should receive a message. This section describes the algorithm used for each exchange type.

2.1.1. Fanout Exchange

The simplest exchange type is a Fanout exchange, which sends each message to every queue bound to the exchange.

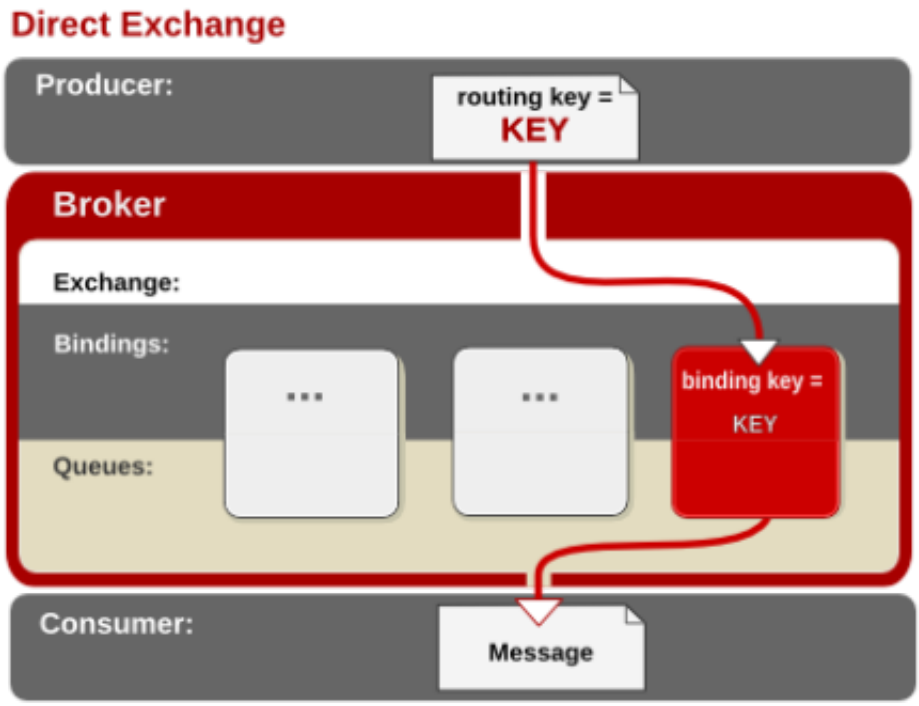


A Fanout exchange sends messages to every queue bound to the exchange.

Every AMQP 0-10 broker contains one predeclared fanout exchange named `amq.fanout`. Additional fanout exchanges can be declared as needed.

2.1.2. Direct Exchange

A message producer can specify a routing key for a message. A routing key is simply a string that indicates a kind of message. In a Direct exchange, a binding specifies a binding key, and an exchange delivers a message to a bound queue if the message's routing key is identical to the queue's binding key.



A Direct exchange sends a message to a queue if the message's routing key is identical to the binding key for the queue.

Every AMQP 0-10 broker contains one predeclared direct exchange named `amq.direct`. Additional direct exchanges can be declared as needed.

2.1.3. Default Exchange

A default exchange uses the name of a queue as the binding key. If a message producer sends a message to the default exchange, and the routing key in the message is the name of a queue, the message is delivered to that queue. In the AMQP 0-10 model, every queue is automatically bound to the default exchange.

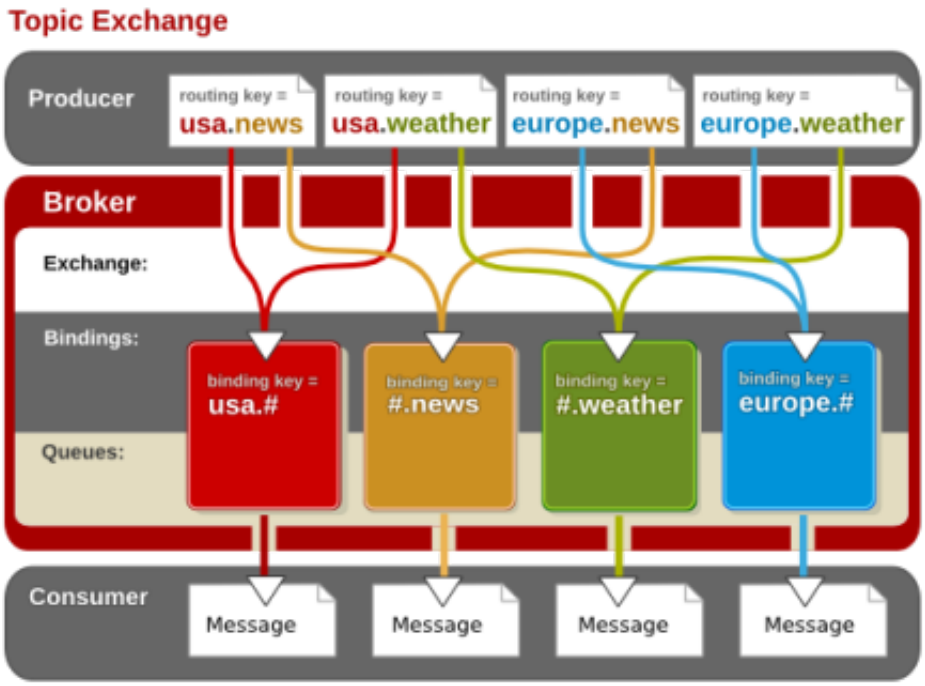
In AMQP 0-10, every exchange except the default exchange has a name. If a message is sent to a messaging broker without specifying an exchange name, it is sent to the default exchange. Every AMQP 0-10 broker has one predeclared default exchange, which has no name. It is not possible to declare additional default exchanges.

2.1.4. Topic Exchange

A Topic exchange is similar to a Direct exchange, but uses keys that contain multiple words separated by a “.” delimiter. A message producer might create messages with routing keys like **`usa.news`**, **`usa.weather`**, **`europe.news`**, and **`europe.weather`**.

Binding keys for a Topic exchanges can include wildcard characters: a “#” matches one or more words, a “*” matches a single word. Typical bindings use binding keys like **`#.news`** (all news items), **`usa.#`** (all items in the USA), or **`usa.weather`** (all USA weather items).

The exchange routes messages to the relevant queue or queues, depending on matches between the routing and binding keys.



A Topic exchange can use multi-part routing keys and bindings that include wildcards. A topic exchange sends a message to a queue if the message's routing key matches the binding key for the queue, using wildcard matching.

> Example 2.1. Using multi-part keys with a topic exchange
> This example demonstrates the use of multi-part keys in a topic exchange.
>
> Suppose a message producer creates four messages concerning news and weather in the United States of America and Europe. The producer creates four different routing keys for the messages, each of which contains two parts. The two parts of the routing keys are separated with a `.` (period) character:
> 1. `usa.news`
> 2. `usa.weather`
> 3. `europe.news`
> 4. `europe.weather`
>
> There are currently four queues bound to the topic exchange. The four queues collect information on:
> 1. Everything related to the USA
> 2. All news
> 3. All weather
> 4. Everything related to Europe
>
> These are defined as a two-part binding key, with the # (pound) character as a wildcard:
> 1. `usa.#`
> 2. `#.news`
> 3. `#.weather`

4. *europe.#*

In this example, the message with the routing key of *usa.weather* will be delivered to two queues - *usa.#* and *#.weather*. Similarly, the message with the routing key of *europe.news* will be delivered to the queues with the binding keys *europe.#* and *#.news*

Every AMQP 0-10 broker contains one predeclared topic exchange named `amq.topic`. Additional topic exchanges can be declared as needed.

2.1.5. Headers Exchange

AMQP 0-10 provides a headers exchange, which allows simple queries on properties in the headers of a message. The binding contains a list of header names and properties, and an argument specifying whether 'all' semantics are used (all header fields must match in order for the message to match) or 'any' semantics (if any header field matches, the message satisfies the binding.

Every AMQP 0-10 broker contains one predeclared headers exchange named `amq.match`. Additional header exchanges can be declared as needed.

2.1.6. Custom Exchange Types

AMQP allows implementations to provide exchange types that are not defined in the standard. These exchange types are referred to as custom exchange types. Custom exchanges are not predeclared.

MRG Messaging provides an XML Exchange, which can route XML messages based on their content. The bindings for an XML Exchange use an XQuery, which is applied to the content and headers of each message to determine whether the message should be routed.

Warning

If the message is non-routable, and if no alternate exchange has been specified, then the message will be dropped. This will result in data loss if these conditions occur.

For instance, suppose the messages sent to an XML exchange contain weather reports like this one:

```
<weather>
    <station>Raleigh-Durham International Airport (KRDU)</station>
    <wind_speed_mph>16</wind_speed_mph>
    <temperature_f>70</temperature_f>
    <dewpoint>35</dewpoint>
</weather>
```

An XML Exchange binding specifies a binding key and an XQuery. A message matches the binding if its routing key matches the binding key and the XQuery evaluates to true.

The XQuery used in an XML Exchange binding can specify a set of conditions based on the structure of the XML. For instance, the following binding matches messages for Raleigh-Durham, with a temperature greater than 50 degrees, a temperature comfortably above the dewpoint, and wind speeds within a range suitable for small sailboats:

```
let $w := ./weather
   return $w/station = 'Raleigh-Durham International Airport (KRDU)'
```

```
      and $w/temperature_f > 50
      and $w/temperature_f - $w/dewpoint > 5
      and $w/wind_speed_mph > 7
      and $w/wind_speed_mph < 20
```

The XML Exchange can also query headers. Each header is bound to an external XQuery variable that has the same name as the header. The query for a binding must declare this external variable in order to use it. A single query may contain conditions on both headers and XML content.

The following query matches a message if the header `control` contains the value 'end', or if the XML content specifies an odd numbered message ID:

```
declare variable $control external;
./message/id mod 2 = 1 or $control = 'end'
```

2.2. Declaring Exchanges

An application can declare new exchanges, which may have either built-in or custom exchange types. A declared exchange continues to exist until the broker terminates or it is explicitly removed; if an exchange is declared to be persistent, it continues to exist even after a server restart.

Because there are no predeclared exchanges for custom exchange types, they can only be used if they are explicitly declared. There are predeclared exchanges for all built-in exchange types except for the default exchange, but there are often good reasons for creating new exchanges for these types as well; for instance:

- *Meaningful names*: Exchanges can be created to provide names that are meaningful in the application domain. For instance, an exchange name can reflect the kind of messages that are sent to the exchange (e.g. “invoices”), geographical regions (e.g. “carolinas”), work queues (e.g. “administrator”), or other names that are meaningful within a given application domain.
- *Organizing the application*: Exchanges can be created to organize the application. For instance, exchanges may be created to reflect priority levels, federation strategies, or other meaningful aspects of application organization.
- *Improving performance for large numbers of bindings*: Exchanges with very large numbers of bindings perform more slowly than exchanges with smaller numbers of bindings. Using a larger number of exchanges reduces the number of bindings on each exchange, and can improve performance.

If an attempt is made to create an exchange that exists, no error is raised, provided the exchange is of the right exchange type. When an exchange is created, the following parameters can be provided to configure the exchange:

Exchange Declaration Parameters

- `type` (string)

  The type of the exchange. May be a built-in type (e.g. “direct”, “fanout”, “topic”) or custom type (e.g. “xml”).

- `alternate_exchange (string)`

  Exchange name for unroutable messages. If a message can not be routed to a queue, it is sent to the alternate exchange if the message's `accept mode` was set to “none”, `discard-unroutable` was set to “false”, and no queue binding is satisfied by the message.

- `passive` (Boolean)

  If set to “true”, test whether the exchange exists, without creating the exchange.

- `durable` (Boolean)

  If set to “true”, the exchange is durable. Durable exchanges remain active if the server is restarted. If an exchange is not durable, it is purged when the server restarts.

Warning

The purge of non-durable exchanges will result in data-loss in exchanges with `durable` not set to “true”.

- `arguments` (FieldTable)

  Implementation-defined options for queue creation. Apache Qpid defines the following options:

  - `qpid.msg_sequence` (Boolean)

    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 sent to the exchange is 0, it is incremented sequentially for each subsequent message.

  - `qpid.ive` (Boolean)

    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.

*Section 12.1, “Using **`qpid-config`**”* shows how to create or delete exchanges using **`qpid-config`**.

When necessary, exchanges can be created on-demand by applications. This is done by specifying a create policy for the address from which a sender and/or receiver is created. For details on addresses and programming applications, see the guide entitled *Programming in Apache Qpid*.

2.3. Exclusive Bindings for Direct Exchanges

In MRG Messaging, direct exchanges support *exclusive bindings*, which ensure that a given binding key is associated with only one queue. If a new binding is created using the same binding key as an existing binding, the new binding is created and the existing binding is deleted. This ensures that a given message is routed to exactly one queue if the routing key of the message matches a binding key.

An exclusive binding is created by adding a key of `qpid.exclusive-binding` to the arguments field of the exchange-bind call

Queues

Queues are bound to one or more exchanges. Messages that are published to the exchanges are routed to queues where the routing and binding keys match. Queues then store messages until they are consumed by clients.

Every queue is bound to the default exchange, which provides a simple and direct method of publishing messages. Other methods are discussed in *Chapter 2, Exchanges*.

Clients receive messages by subscribing to the queue that contains the messages they want to see. These subscribers can browse through messages without acquiring them, leaving messages on the queue for other subscribers to browse. Alternatively, clients can consume the messages, permanently removing them from the queue once they are read. This creates competition for messages. Once they have been removed from the queue, they are no longer available for other consumers to read.

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.

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.

Deleting Queues

When a queue is deleted the queue and any messages it contains are destroyed, and all bindings that refer to the queue are removed. When the broker receives a queue delete command for a particular queue, the following checks are made before the deletion occurs:

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

Once the queue has been deleted, the management object associated with the queue will remain. This makes it possible to see the deleted queue using **`qpid-tool`** or the MRG Management Console. These tools will show the queue with a timestamp of when it was deleted.

Automatically deleted queues

When a queue is created, its lifecycle can be limited by marking it to be automatically deleted. This can be achieved by setting the *`auto-delete`* field. Auto-delete is handled differently for exclusive and non-exclusive queues:

- An exclusive, auto-deleted queue is deleted when the session that declared it ends

- A non-exclusive auto-deleted queue will be deleted once the last subscriber is canceled. It will not not be deleted until at least one session has subscribed and then canceled that subscription.

Rejected and orphaned messages

Messages can be *rejected* by a subscribed client. Once a message has been rejected by a client, it will not be re-delivered. Messages can also be *orphaned* if they are left on a queue when that queue is deleted.

For both rejected and orphaned messages, the system can be configured to handle them using an *alternate-exchange*. An alternate exchange is specified when the queue is declared. Any rejected or orphaned messages will automatically be routed to the alternate exchange, to be re-routed to other bound queues or deleted if necessary. If no alternate exchange is specified, all rejected and orphaned messages will be automatically deleted.

Controlling queue size

A size limit can be set on a queue by specifying values for **qpid.max_count** and **qpid.max_size** when declaring the queue. By default, an exception will be raised when published messages exceed this limit.

The default behavior can be controlled by changing the **qpid.policy_type** option. The possible values for this option are:

reject
: The publisher of a message that exceeds the limit receives an exception. This is the default behavior for all non-durable queues

flow_to_disk
: The content of messages that exceed the limit is freed from memory and held on disk. This occurs for both persistent and non-persistent messages and is the default behavior for durable queues

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.

Queue Threshold Alert Configuration

Steadily increasing queue depth is a common symptom of problems in a MRG Messaging application. The queue may run out of capacity or the broker may run out of resources if this condition is not rectified. Queues can be configured to raise alerts when depth reaches pre-configured thresholds so that administrators can take corrective action or be alerted to the situation before it becomes more critical.

When queue threshold alerts are generated can be configured using arguments for alert count and alert size. An alert is issued when a message is added to the queue and the size or count becomes equal or greater than the configured limits. By default the threshold alert value is 80%, an alert is generated when 80% of any configured queue limit is reached. The ratio used can be configured via the broker option **--default-event-threshold-ratio** and setting it to zero disables alerts of thresholds from capacity. If alerts for both message count and size are configured then a single alert will trigger for the condition that occurs first.

The alert threshold may also be explicitly defined per queue by specifying **qpid.alert_count** and/or **qpid.alert_size** in the queue arguments when creating the queue.

Alerts are sent out as map messages, they can be subscribed to and received using the regular MRG Messaging API. This is shown in following python code example:

```
conn = Connection.establish("localhost:5672")
session = conn.session()
rcv = session.receiver("qmf.default.topic/
agent.ind.event.org_apache_qpid_broker.queueThresholdExceeded.#")
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()
```

To avoid excess traffic, once an alert for a queue has been issued there will be no further alerts for a configured period. This period can be controlled per queue using the `qpid.alert_repeat_gap` argument to specify the minimum interval between events in seconds. The default value is 60 seconds.

Important

Messages issued during the alert repeat gap will not trigger an alert after the alert repeat gap time has elapsed. There is no queue for messages, a new alert will only be triggered by a queue state change after the alert repeat gap.

Note

To comply with argument names already supported, the following aliases for arguments used above are also recognized:

- `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`

Ignoring locally published messages

Under the AMQP model, exclusive, auto-deleted queues are often bound to an exchange that enables the queue owner to subscribe to messages for consumption. In this case, a queue owner might send itself messages using that exchange, but have no need to receive those messages.

To ignore locally published messages, a *`no-local`* key can be specified in the arguments to the declare used to create the queue. The value of this key is irrelevant, its presence alone will cause the correct behavior. This key will cause the queue to discard any messages that were published by the same connection as that of the session that owns the queue.

Last value queues

The *last value* queue type causes logically updated versions of previous messages to appear to overwrite the older messages.

The last value queue uses the value of the *qpid.LVQ_key* to determine whether a newly published message is an update to an existing message on the queue. If this is the case, the new message will appear to overwrite the older message. A subscriber that requests messages after this has occurred will see only the newer message.

There are two types of Last Value Queue:

- **LVQ**
- **LVQ NO BROWSE**

LVQ uses a header as a key. If the key matches it replaces the message in the queue, unless:
- the message with the matching key has been acquired
- the message with the matching key has been browsed

In these cases the message is placed into the queue in FIFO. If another message with the same key is received the message that has not been accessed will be replaced. These two exceptions protect the consumer from missing the last update if a consumer or browser has accessed a message.

LVQ NO BROWSE also uses a header for a key. If the key matches it replaces the message in the queue unless the message with the matching key has been acquired. In this case browsed messaged are not invalidated, so updates to messages already browsed on a key will be missed. If a new subscription is created the latest values will be seen.

To use this feature, add a *qpid.last_value_queue* or *qpid.last_value_queue_no_browse* key to the arguments of queue declare. The value of the key is user-selected and used only for key matching. Messages published to the queue then need to specify a value for the *qpid.LVQ_key* in the headers of messages they publish.

Priority Queuing Configuration

To enable the support for message priority feature a value needs to be given for *qpid.priorities*. This value is given in the arguments to the declare that creates the queue. The value determines the number of distinct priority levels recognized by the queue (up to a maximum of 10). The default is 1 level (no prioritized delivery). For example the declaration for a queue with the address *my-queue* and priority level 10 would be:

```
my-queue; {create: always, node:{x-declare:{arguments:{qpid.priorities:10}}}}
```

For level values greater than 1 but less than 10, specific priorities are mapped to levels as defined in the AMQP 0-10 specification (refer to rule 'priority-level-implementation'). For 10 levels the priorities map directly to levels.

Ring queues with more than one priority level will remove messages by priority and then age. This results in the lowest priority messages being removed first, and within a given priority level the oldest message will be removed first.

The delivery order for browsing subscribers should be considered undefined on a queue with multiple priority levels. Presently they are browsed in FIFO order, but this should not be relied upon.

Durable queues

Queues can be defined as *durable*. A durable queue is stored in memory, and can survive a restart of the broker. However, messages on the queue must also be declared as persistent for them to be recovered..

There are other options that can be used with durable queues to control the sizing and tuning of the journal used to record queue state on disk. For more information, see *Chapter 6, Persistence*.

Enforcing persistence on the last node in a cluster

PersistLastNode is used if a cluster fails down to a single node. In this situation, a queue would treat all transient messages as persistent until additional nodes in the cluster are restored.

This mode will not be triggered if a cluster is started with only one node. It will only be triggered if active nodes fail until there is only one node remaining.

If this mode is used, queues must be configured to be durable, otherwise it will fail to persist.

Configuring queue options

This section explains how to set queue options from the shell prompt using the **qpid-config** tool.

Note

Information on obtaining and using the **qpid-config** tool is in *Section 12.1, "Using **qpid-config**"*.

1. List information on all existing queues by using the **queues** command:

```
$ qpid-config  queues

         Store Size
Durable  AutoDel  Excl  Bindings  (files x file pages)  Queue Name
================================================================================
 N    N      N         1                          pub_start
 N         N      N         1                          pub_done
 N         N      N         1                          sub_ready
 N         N      N         1                          sub_done
 N         N      N         1                          perftest0
 N         Y      N         2                          mgmt-3206ff16-fb29-4a30-82ea
 N         Y      N         2                          repl-3206ff16-fb29-4a30-82ea
 N         Y      N         2                          mgmt-df06c7a6-4ce7-426a-9f66
 N         Y      N         2                          repl-df06c7a6-4ce7-426a-9f66
```

2. Queues are created using a command with this syntax:

```
$ qpid-config [options] add queue queue_name [add queue options]
```

The possible options are:

--durable
Makes the queue durable (see *Chapter 6, Persistence* for more information about durable queues

--cluster-durable
Makes the queue durable if there is only one functioning cluster node

--file-count *NUMBER*
Set the number of files in the persistence journal for the queue. Defaults to 8

`--file-size` *`NUMBER`*
Set the number of pages in the file (each page is 64KB). Defaults to 24

`--max-queue-size` *`NUMBER`*
Maximum queue size in bytes.

`--max-queue-count` *`NUMBER`*
Maximum queue size in number of messages

`--policy-type` *`TYPE`*
Action to take when queue limit is reached. *TYPE* can be:
- *`reject`*
- *`flow_to_disk`*
- *`ring`*
- *`ring_strict`*

`--last-value-queue`
Enable last value queue behavior on the queue

3. To delete a queue, use the **`del queue`** command with the name of the queue to remove:

```
$ qpid-config del queue queue_name
```

Creating queues from within applications

Applications can also create queues dynamically when required. See the guide entitled *Programming with Apache Qpid* for further details.

3.1. Producer Flow Control

The broker supports using flow control to block message producers that are at risk of overflowing a destination queue. The blocked message producer will become unblocked when enough of messages are delivered and acknowledged.

Each queue in the broker has two threshold values associated with it:

- **`flow_stop_threshold`** - the queue resource utilization level that enables flow control if the queue exceeds. Once crossed, the queue is considered in danger of overflow.
- **`flow_resume_threshold`** - the queue resource utilization level that disables flow control if the queue drops below. Once crossed, the queue is no longer considered in danger of overflow.

Queue resource utilization is the total number of messages currently enqueued, or the total sum of all message content in bytes.

Important

The value for a queue's **`flow_stop_threshold`** must be greater than or equal to the value of the queue's **`flow_resume_threshold`**.

Flow Threshold Configuration

By default, the broker assigns the queue's flow stop and resume thresholds when it is created. The broker also allows a user to manually configure the flow control thresholds on a per queue basis.

Queues that have been configured with a Limit Policy of type *`ring`* or *`ring-strict`* do NOT have queue flow thresholds enabled.

The flow control state of a queue can be determined by the **`flowState`** boolean in the queue's QMF management object. The queue's management object also contains a counter that increments each time flow control becomes active for the queue.

The broker applies a threshold ratio to compute a queue's default flow control configuration. These thresholds are expressed as a percentage of a queue's maximum capacity. One value is to determine the stop threshold, and the other for the resume threshold. You can configure these percentages using the following broker configuration options:

```
--default-flow-stop-threshold=flow control activated capacity level
--default-flow-resume-threshold=flow control activated capacity level
```

An example of setting the default flow stop threshold to 90% of a queue's maximum capacity and the flow resume threshold to 75% of the maximum capacity:

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

If a queue is created with a **`default-queue-limit`** of 10000 bytes, then the default flow stop threshold would be 90% of 10000 = 9000 bytes and the flow resume threshold would be 75% of 10000 = 7500. The same computation is performed should a queue be created with a maximum size expressed as a message count instead of a byte count.

The default value of **`default-flow-stop-threshold`** is 80% and **`default-flow-resume-threshold`** is 70%. Disable the default queue flow control broker-wide by assigning the value zero to configuration variables. Note that flow control may still be applied manually on a per-queue basis in this case.

Flow thresholds can be set in the **`queue.declare`** method, via the **`arguments`** parameter map. The following keys can be used in the **`arguments`** map for setting flow thresholds:

Table 3.1. **`queue.declare`** Flow Control Arguments

`queue.declare` Flow Control 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.

Note

To disable flow control on a per queue basis, set the **`flow-stop-size`** and **`flow-stop-count`** to zero for the queue.

You can manually set flow thresholds when creating a queue. The following options may be provided when adding a queue using the **qpid-config** command line tool:

```
--flow-stop-size=N
--flow-resume-size=N
--flow-stop-count=N
--flow-resume-count=N
```

The value of *N* in the size type thresholds is the flow stop threshold in bytes, for the count type thresholds it is the total number of messages.

Queue Threshold Example

Consider a queue with a maximum limit of 1000 on the total number of messages that may be enqueued to that queue. If a user configures **flow-stop-count** with the value 900, and **flow_resume_count** with the value 500. With these thresholds the following queue activity occurs:

- When the total number of enqueued messages is less than or equal to 900, the queue's flow control state is OFF.
- When the total number of enqueued messages is greater than 900, the queue's flow control state transitions to ON.
- When the queue's flow control state is "ON", it remains "ON" until the total number of enqueued messages is less than 500. At that point, the queue's flow control state transitions to "OFF".

Important

Thresholds can be set using both total message counts and total byte counts. In these cases, the following applies:

- Flow control is "ON" when either stop threshold value is crossed.
- Flow control remains "ON" until both resume thresholds are satisfied.

Sessions

Sessions are a uniquely identified conversation between a client and the broker. Multiple distinct sessions can share the same connection to the broker. An application process can also have multiple connections open to a broker.

Completion of commands issued by a client

All the interaction with a broker is done by issuing commands on a session. The valid commands are defined by the AMQP specification and include operations for declaring a queue and transferring a message. The broker can also be asked to indicate when it completes commands. This allows clients to confirm successful execution.

Subscriptions and received messages

To receive messages from the broker, the client subscribes to a queue. The broker will then deliver available messages for that subscription.

Message acquisition and acceptance

Message acquisition and acceptance occurs as a transfer of ownership of a message.

By *acquiring* a message, a subscriber is given the option to take ownership of that message. A message can be acquired by only one subscriber at any time. While a message is acquired by a subscriber, the broker can not give it to any other. The subscriber can then confirm that it wishes to accept ownership of any acquired message by *accepting* that message. At this point the broker relinquishes ownership and permanently removes the message from the queue.

A subscriber might choose not to take ownership of an acquired message. In this case, the message is *released*. This allows the broker to re-deliver the message to any other available subscriber - this can include the subscriber that just released the message.

Acquired messages can also be *rejected*. This tells the broker that the message is not valid for further delivery and should be *dead-lettered* or discarded. See *Rejected and orphaned messages* for more information on rejected messages.

The default acquire mode for a subscription is the `pre-acquired` mode. In this mode, delivered messages are implicitly acquired by the subscriber that receives them. Alternatively, a subscriber can use the `not-acquired` mode. This allows the subscriber to request that it is sent messages that are on the queue without acquiring them. Messages can then be acquired explicitly.

In addition to the acquire mode, a subscription can also set an accept mode. In the `explicit` mode, ownership of a message is transferred to the acquiring subscriber only when the message has been explicitly accepted. Alternatively, if the accept mode is set to *none*, messages are considered accepted as soon as they are acquired. This mode is less reliable, but is often suitable where message loss on session failure poses no risk to the system.

Flow control and completion of messages sent to a subscriber

A subscriber can control the flow of messages from a subscribed queue by allocating *credit* to the broker for a particular number of messages or a total size of message content. As the broker delivers messages it spends this credit by decrementing the message credit by one and decrementing the size credit by the size of the content of the message. The broker cannot send a message to a subscription for which it does not have sufficient credit.

There are two modes of credit allocation defined by the AMQP specification:

- In *`credit`* mode, credit must be explicitly re-issued by the subscriber before the broker can recommence sending messages
- In *`window`* mode, the credit is automatically reissued for received messages. In this mode, the client indicates that a message has been received by informing the broker of the completion of the transfer. Though completion is essentially a form of acknowledgment, it should not be confused with acceptance which is an confirmation of ownership transfer.

In both modes, unlimited credit can be allocated for the message count and the total content size. MRG Messaging also supports producer flow control, see *Section 3.1, “Producer Flow Control ”* for more information.

Transactions

A *transaction* is an atomic group of published or accepted messages. It can be viewed as a set of enqueue and dequeue operations on one or more queues. *Enqueues* occur when messages are published on one or more queues. *Dequeues* occur when a message is accepted.

The atomicity of the group of operations within a transaction means that they will either all succeed or all fail. In these terms, success indicates that the published messages all become available on their respective queues, and the accepted messages are permanently removed from their respective queues. Failure indicates that the published messages are discarded, the acceptances are ignored and the messages remain unaccepted.

A message published under a transaction will not become available to subscribers on any queue until the transaction is committed. A message that is accepted under a transaction will not be dequeued until the transaction is committed.

If a transaction is *rolled-back* then all messages published under that transaction will be discarded. All messages accepted under it will remain unaccepted. This means that they will not be returned to the queue or re-delivered unless it has been explicitly requested. The API used will determine the expected behavior. If a transactional session ends without committing a transaction, the transaction itself will be automatically rolled-back.

Both local and distributed transactions are supported by **`qpidd`**. In a local transaction the only atomic operations are those that occur on the broker to which the transactional session is connected. Distributed transactions use *two-phase commit* to achieve atomicity across multiple services.

Persistence

AMQP is designed to send messages reliably. If any component of the messaging system fails, no message will be lost. The AMQP protocol provides a model for acknowledging message publication and message consumption of messages, and this model clearly allocates responsibility for each message should a network failure, client failure, or broker failure occur. However, if the broker fails, this model only guarantees delivery if message persistence is used to ensure that messages can be recovered when the broker is restarted. Message persistence writes messages to a journal, where they can be retrieved if the broker fails. For speed, the journal is written using AIO (Asynchronous Input/Output).

In addition to guaranteed delivery, persistence allows configurations to survive when the broker is shut down—exchanges and queues that are declared durable will be restored, along with their bindings, when the broker is started again. This configuration information is stored using *Berkeley DB*[1].

6.1. Persistent Messages, Durable Queues, and Guaranteed Delivery

In MRG Messaging, a persistent message is a message that must not be lost, even if the broker fails. A *durable* is saved on disk when its server shuts down or fails, and then reestablished when the server re-starts. A *persistent* queue is able to write messages to disk so that they will not be lost in the event of server shut down or failure. In Red Hat Enterprise MRG these two queue properties currently always go together. Queues with these properties are referred to as **`durable`**. Note that durable queues do not persist all messages. The message itself must also be marked persistent, or even a durable queue will handle it normally. A queue must be declared durable at the time it is created.

In C++, the following code creates a durable queue:

```
session.queueDeclare(arg::queue=name, arg::durable=true);
```

Note

If a queue is declared **`autoDelete`**, it is deleted when the last client unsubscribes to the queue or terminates. This is true even if the queue is durable.

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

[1] http://www.oracle.com/technology/products/berkeley-db/index.html

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 durable, it is not written to disk even if it is on a durable queue.

Table 6.1. 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.

6.2. Running the MRG Messaging Broker with Persistence

The MRG 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:

```
2009-nov-04 20:39:58 notice Journal "TplStore": Created
2009-nov-04 20:39:58 notice Store module initialized; dir=/home/aperson/.qpidd
```

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 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. See *Table 6.2, "Persistence Options"* for options that change this behavior. 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.*

6.3. Configuring the Journal

MRG 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.

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.

When you create a queue using **qpid-config**, you can set the size of the journal using the **--file-count** and **--file-size** options. More information on these commands can be found in *Configuring queue options*.

In C++, you can set the file count and file size by specifying `qpid.file_size` and `qpid.file_count` in `Session.createQueue()`:

```
FieldTable journal_settings;
journal_settings.setInt("qpid.file_size", 20);
journal_settings.setInt("qpid.file_count", 12);

session.queueDeclare(arg::queue="my_queue", arg::durable=true,
 arg::arguments=journal_settings);
```

In Python, you can set the file count and file size by specifying `qpid.file_size` and `qpid.file_count` in `session.queue_declare()`:

```
session.queue_declare(queue="my_queue", durable=True, arguments={"qpid.file_size":20,
 "qpid.file_count":12})
```

The following table lists the options that can be set for persistence when starting the MRG Messaging broker.

Table 6.2. Persistence Options

Persistence Options	
--store-dir *DIRECTORY*	The 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 of each journal is **num-jfiles** * **jfile-size-pgs**.
--jfile-size-pgs *NUMBER*	The size of each journal file, in multiples of 64KB. The default is 24. Minimum is 1, maximum is 32768. The total size of each journal is **num-jfiles** * **jfile-size-pgs**. 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 will 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 32768.
--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 will 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.

Correcting Illegal Numeric Parameters

If an out-of-range or illegal parameter is supplied to the store, it will automatically replace it with the closest legal value and place a warning in the log file rather than fail. This applies to all numeric store parameters.

For example, starting the store with **`--num-jfiles 1`** (which is out-of-range, the minimum allowed is 4), the store will automatically substitute a value of 4 and place 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.

6.4. Determining Journal Size

Section 6.3, "Configuring the Journal" explains how to set the size of a journal for a persistent queue. 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 some sort of load balancing that matches the rate of publishing with the rate of consuming.

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.

6.4.1. 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.

6.4.2. 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.

6.4.3. Calculating 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 6.1. Calculating 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.
>
> Let's 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

6.4.4. Calculating Journal Size (With Transactions)

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.

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:

Message enqueue 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 6.2. 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.

Let's 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

6.4.5. Resizing 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.

High Availability Messaging Clusters

High Availability Messaging Clusters provide fault tolerance by ensuring that every broker in a *cluster* has the same queues, exchanges, messages, and bindings, and allowing a client to *fail over* to a new broker and continue without any loss of messages if the current broker fails or becomes unavailable. Because all brokers are automatically kept in a consistent state, clients can connect to and use any broker in a cluster. 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.

High Availability Messaging Clusters are implemented using using the *OpenAIS Cluster Framework*[1]. See the *MRG Messaging Installation Guide* for instructions on installing OpenAIS to enable clustering for qpidd.

An OpenAIS daemon runs on every machine in the cluster, and these daemons communicate using multicast on a particular address. Every qpidd process in a cluster joins a named group that is automatically synchronized using OpenAIS Closed Process Groups (CPG) — the qpidd processes multicast events to the named group, and CPG ensures that each qpidd process receives all the events in the same sequence. All members get an identical sequence of events, so they can all update their state consistently.

Two messaging brokers are in the same cluster if

1. They run on hosts in the same OpenAIS cluster; that is, OpenAIS is configured with the same mcastaddr, mcastport and bindnetaddr, and

2. They use the same cluster name.

High Availability Clustering has a cost: in order to allow each broker in a cluster to continue the work of any other broker, a cluster must replicate state for all brokers in the cluster. Because of this, the brokers in a cluster should normally be on a LAN; there should be fast and reliable connections between brokers. Even on a LAN, using multiple brokers in a cluster is somewhat slower than using a single broker without clustering. This may be counter-intuitive for people who are used to clustering in the context of High Performance Computing or High Throughput Computing, where clustering increases performance or throughput.

High Availability Messaging Clusters should be used together with Red Hat Clustering Services (RHCS); without RHCS, clusters are vulnerable to the "split-brain" condition, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. See the documentation on the **`--cluster-cman`** option for details on running using RHCS with High Availability Messaging Clusters. See the *CMAN documentation*[2] for more detail on CMAN and split-brain conditions. Use the **`--cluster-cman`** option to enable RHCS when starting the broker.

7.1. Starting a Broker in a Cluster

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="local_test_cluster"
```

[1] http://www.openais.org/
[2] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Cluster_Suite_Overview/s1-hasci-overview-CSO.html#s2-clumembership-overview-CSO

On RHEL6, you must 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
}
```

On RHEL5, 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, you must ensure that the primary group for the user running qpidd is ais. You can set the primary group using **newgrp**:

```
$ newgrp ais
```

You can then run the broker from the command line, specifying the cluster name as an option.

```
[jonathan@localhost]$ qpidd --cluster-name="local_test_cluster"
```

All brokers in a cluster must have identical configuration, with a few exceptions noted below. They must load the same set of plug-ins, and have matching configuration files and command line arguments. They should also have identical ACL files and SASL databases if these are used. 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.

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.

The qpid log contains entries that record significant clustering events, e.g. when a broker becomes a member of a cluster, the membership of a cluster is changed, or an old journal is moved out of the way. For instance, the following message states that a broker has been added to a cluster as the first node:

```
2009-07-09 18:13:41 info 127.0.0.1:1410(READY) member update: 127.0.0.1:1410(member)
2009-07-09 18:13:41 notice 127.0.0.1:1410(READY) first in cluster
```

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.

The following options are available for clustering:

Table 7.1. Options for High Availability Messaging Cluster

Options for High Availability Messaging Clusters	
--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 if 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

Options for High Availability Messaging Clusters	
	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*[3] 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

If a broker is unable to establish a connection to another broker in the cluster, the log will contain SASL errors, e.g:

```
2009-aug-04 10:17:37 info SASL: Authentication failed: SASL(-13): user not found: Password
 verification failed
```

You can 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. In most environment, it is easiest to create an account with the same user name and password on each broker in the cluster, and use these as the **`cluster-username`** and **`cluster-password`**. You can also set the SASL mode using **`cluster-mechanism`**. Remember that 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.

Once the cluster is running, run **`qpid-cluster`** to make sure that the brokers are running as one cluster. See the following section for details.

If the cluster is correctly configured, queues and messages are replicated to all brokers in the cluster, so an easy way to test the cluster is to 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. The **`drain`** and **`spout`** programs can be used for this test.

7.2. 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
```

Let's 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. Let's use the ID to shut the broker down using the **`--stop`** command:

```
$ ./qpid-cluster localhost:6664 --stop 127.0.0.1:13143
```

7.3. Failover in Clients

If a client is connected to a broker, the connection fails if the broker crashes or is killed. If heartbeat is enabled for the connection, a connection also fails if the broker hangs, the machine the broker is

running on fails, or the network connection to the broker is lost — the connection fails no later than twice the heartbeat interval.

When a client's connection to a broker fails, any sent messages that have been acknowledged to the sender will have been replicated to all brokers in the cluster, any received messages that have not yet been acknowledged by the receiving client requeued to all brokers, and the client API notifies the application of the failure by throwing an exception.

Clients 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.

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. The Java JMS client enables heartbeat by default. See the sections on Failover in Java JMS Clients and Failover and the Qpid Messaging API for the code to enable heartbeat.

7.3.1. Failover in Java JMS Clients

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'
```

This 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.

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
```

7.3.2. Failover and the Qpid Messaging API

The Qpid Messaging API also supports automatic reconnection in the event a connection fails. Senders can also be configured to replay any in-doubt messages (i.e. messages which were sent

but not acknowledged by the broker. See "Connection Options" and "Sender Capacity and Replay" in *Programming in Apache Qpid* 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 via the 'heartbeat' option.

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

7.4. Error handling in Clusters

If a broker crashes or is killed, or a broker machine failure, broker connection failure, or a broker hang is detected, the other brokers in the cluster are notified that it 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 it resumes, the broker 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.

7.5. Persistence in High Availability Message 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 all transient or all persistent members, mixed clusters are not allowed. Each broker in a persistent cluster has it's own independent replica of the cluster's state it its store.

7.5.1. 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 is shutdown by the utility **`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.

7.5.2. 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.

7.5.3. Stopping 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.

7.5.4. Starting a persistent cluster with no clean store

If the cluster has previously had a total failure and there are no clean stores then the brokers will fail to start with the log message **`Cannot recover, no clean store.`** If this happens you can start the cluster by marking one of the stores "clean" as follows:

1. Move the latest store backup into place in the brokers data-directory. The backups end in a 4 digit number, the latest backup is the highest number.

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

2. Mark the store as clean:

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

Now you can start the cluster, all members will be initialized from the store you marked as clean.

Important

`qpid-cluster-store` is intended to be used only in disaster recovery situations when a cluster fails completely with no clean store. It should not be used on a running cluster member.

7.5.5. 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 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.

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 may 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 change, responsibilities change, at different times of day, or to reflect other changing conditions.

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

- Geography: Customer requests may be routed to a processing location close to the customer.
- Service Type: High value customers may be routed to more responsive servers.
- Load balancing: Routing among brokers may be changed dynamically to account for changes in actual or anticipated load.
- High Availability: Routing may be changed to a new broker if an existing broker becomes unavailable.
- WAN Connectivity: Federated routes may 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.

8.1. 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 in order 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, and dynamic exchange routes.

8.1.1. 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.

8.1.2. 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.

8.1.3. Dynamic Exchange Routes

Dynamic exchange routes allow a client to create bindings to an exchange on one broker, and receive messages that satisfy the conditions of these bindings not only from the exchange to which the client created the binding, but also from other exchanges that are connected to it using dynamic exchange routes. If the client modifies the bindings for a given exchange, they are also modified for dynamic exchange routes associated with that exchange.

Dynamic exchange routes apply all the bindings of a destination exchange to a source exchange, so that any message that would match one of these bindings is routed to the destination exchange. If bindings are added or removed from the destination exchange, these changes are reflected in the dynamic exchange route—when the destination broker creates a binding with a given binding key, this is reflected in the route, and when the destination broker drops a binding with a binding key, the route no longer incurs the overhead of transferring messages that match the binding key among brokers. If two exchanges have dynamic exchange routes to each other, then all bindings in each exchange are reflected in the dynamic exchange route of the other. 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.

8.2. 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, you should 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, you should take care to ensure that it is not possible for any messages to cycle between A and B. Looping message traffic can flood the federated network. The topologies discussed above do not have message loops. A ring topology with bidirectional links is one example of a topology that does cause this problem, because a given broker can receive the same message from two different brokers. Mesh topologies can also cause this problem.

8.3. Federation among High Availability Message 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. Because message state is replicated within a cluster, there is little sense defining message routes between brokers in the same cluster. A federated link between clusters will not cause multiple transmission of each message between the source and destination cluster.

To create a message route between two clusters, simply 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.

8.4. 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.

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] list  [<broker>]
qpid-route [OPTIONS] flush [<broker>]
qpid-route [OPTIONS] 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`**.

These are the options for **qpid-route**:

Table 8.1. **qpid-route** options

-v	Verbose output.
-q	Quiet output, will not print duplicate warnings.
-d	Make the route durable.
--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

8.4.1. Creating and Deleting Queue Routes

The syntax for creating and deleting queue routes is as follows:

```
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>
```

For instance, the following creates 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
```

If the **-d** option is specified, this queue route is persistent, and will be restored if one or both of the brokers is restarted:

```
$ qpid-route -d queue add localhost:10001 localhost:10002 amq.fanout public
```

The **del** command takes the same arguments as the **add** command. The following command deletes the queue route described above:

```
$ qpid-route queue del localhost:10001 localhost:10002 amq.fanout public
```

8.4.2. Creating and Deleting Exchange Routes

The syntax for creating and deleting exchange routes is as follows:

```
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>]
```

For instance, the following creates 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.#
```

In many applications, messages published to the destination exchange should also be routed to the source exchange. This is accomplished by creating a second exchange route, reversing the roles of the two exchanges:

```
$ qpid-route route add localhost:10002 localhost:10001 amq.topic global.#
```

If the **`-d`** option is specified, the exchange route is persistent, and will be restored if one or both of the brokers is restarted:

```
$ qpid-route -d route add localhost:10001 localhost:10002 amq.fanout public
```

The **`del`** command takes the same arguments as the **`add`** command. The following command deletes the first exchange route described above:

```
$ qpid-route route del localhost:10001 localhost:10002 amq.topic global.#
```

8.4.3. Deleting all routes for a broker

Use the **`flush`** command to delete all routes for a given broker:

```
qpid-route [OPTIONS] flush [<broker>]
```

For instance, the following command deletes all routes for the broker **`localhost:10001`**:

```
$ qpid-route flush localhost:10001
```

8.4.4. Creating and Deleting Dynamic Exchange Routes

The syntax for creating and deleting dynamic exchange routes is as follows:

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

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

In the following examples, we will route messages from a topic exchange. We will create a new topic exchange and federate it so that we are not affected by other all clients that use the built-in **amq.topic** exchange. The following commands 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
```

Now let's 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** if they match any binding on the destination broker's **fed.topic** exchange:

```
$ 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.

In many applications, messages published to the destination exchange should also be routed to the source exchange. This is accomplished by creating a second dynamic exchange route, reversing the roles of the two exchanges:

```
$ qpid-route dynamic add localhost:10004 localhost:10003 fed.topic
```

If the **-d** option is specified, the exchange route is persistent, and 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.

The **del** command takes the same arguments as the **add** command. The following command deletes 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.

8.4.5. Viewing Routes

The **route list** command shows the routes associated with an individual broker. For instance, suppose we have created the following two routes:

```
$ qpid-route dynamic add localhost:10003 localhost:10004 fed.topic
$ qpid-route dynamic add localhost:10004 localhost:10003 fed.topic
```

We can now use **`route list`** to show all routes for the broker **`localhost:10003`**:

```
$ qpid-route route list localhost:10003
localhost:10003 localhost:10004 fed.topic <dynamic>
```

Note that this shows only one of the two routes we created, the route for which **`localhost:10003`** is a destination. If we want to see the route for which **`localhost:10004`** is a destination, we need to do another route list:

```
$ qpid-route route list localhost:10004
localhost:10004 localhost:10003 fed.topic <dynamic>
```

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 instance, here is the output 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 particularly helpful for larger, more complex networks. Let's configure a somewhat more complex 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
```

Now we can use **`route map`** starting with any one broker, and 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
```

8.4.6. Resilient Connections

When a broker route is created, or when a durable broker route is restored after broker restart, a connection is created between the source broker and the destination broker. The connections used between brokers are called *resilient connections*; 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, and continues to retry every 64 seconds thereafter. If the connection fails due to an authentication problem, it will not continue to retry.

The command **`list connections`** can be used to show the resilient connections for a broker:

```
$ qpid-route list connections 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 8.2. State values in **`$ qpid-route list connections`**

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.

Replicated Queues

A *replicated queue* is a queue that generates notification events when a message is enqueued or dequeued, so that its state can be replicated. 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* 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.

It is useful to note that the replication mechanism will behave as a load balancer when more than one backup broker is configured for a replicated queue.

Queue replication is a high availability feature. Unlike High Availability Messaging Clusters, queue replication does not slow broker performance, and works well across a Wide Area Network.

9.1. Configuring Queue Replication

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.

9.1.1. Configuring Queue Replication on the Source Broker

To configure queue replication on the source broker, you enable a replication plugin, specify a replication event queue when the broker starts, and create one or more replicating queues. This section shows how to perform these steps.

1. When you start the MRG Messaging broker, specify the name of the replication event queue using the **`--replication-queue`** option. This can be the name of an existing durable queue created for the purpose, or the **`--create-replication-queue`** option can be used to indicate that a new transient queue should be created with the specified name. Add these options to **`qpidd.conf`** by creating a replication event queue named **`my_repl_queue`** when the broker starts.

   ```
   replication-queue=my_repl_queue
   create-replication-queue=true
   ```

2. Start the source broker:

   ```
   $ sudo service qpidd start
   ```

3. Create one or more replicated queues. If you create a queue using the **`qpid-config`** command, use the **`--generate-queue-events`** option with the value **`2`** to replicate both enqueues and dequeues, or the value **`1`** to replicate only enqueues:

   ```
   ```

```
$ qpid-config --broker-addr src-host add queue my_repl_queue --generate-queue-events 2
```

If the queue is created within a program, set the queue options to enable queue events, using the key **`qpid.queue_event_generation`**, and the value **`1`** to replicate only enqueue events, or **`2`** to replicate both enqueue and dequeue events. For instance, in C++ this can be done with the following code:

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

At this point, the source broker has been configured for queue replication. We still need to configure the backup broker and create a federated message route between the two brokers.

9.1.2. Configuring Queue Replication on the Backup 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, and create a backup queue that corresponds to each replicated queue.

1. On the backup broker, configure **`qpidd.conf`** to load the **`replication_exchange.so`** plugin. You can do this with the following line:

   ```
   load-module="replication_exchange.so"
   ```

2. Start the backup broker:

   ```
   $ sudo service qpidd start
   ```

3. Create a replication exchange on the backup broker, specifying address of the backup host (in this example, "backup-host"), the type of the exchange ("replication"), and the name of the newly created replication exchange ("replication-exchange").

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

4. Create a queue on the backup broker with the same name as the replicated queue on the source broker.

   ```
   $ qpid-config --broker-addr backup-host add queue my_repl_queue --generate-queue-events 2
   ```

At this point, the source broker and the backup broker have been configured for queue replication. We still to create a federated message route between the two brokers.

9.1.3. Creating a Message Route from Source Broker to Backup Broker

Once the source broker and the backup broker are configured, queue replication can be activated by creating a federated message route between the replication event queue ("replication-event-queue") on the source broker (running on the host "src-host") and the replication exchange ("replication-exchange") on the backup broker (running on the host ("backup-host"), using broker federation. To ensure that replication events are not lost if the route fails, turn on acknowledgment. We will batch acknowledgments using a batch size of 50 for the sake of efficiency:

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

At this point, queue replication should be functioning.

9.2. Using Backup Queues in Messaging Clients

If messaging clients connect to a backup queue and perform operations that change the state of the queue, it will no longer represent the state of the replicated queue. If clients are 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.

9.3. Replicated Queues and High Availability Messaging Clusters

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

1. The members of each cluster are communicated at the time the messaging route is established. If the source broker goes down or 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.

2. 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.

Security

This chapter describes how authentication, rule-based authorization, encryption, and digital signing can be accomplished using MRG Messaging. Authentication is the process of verifying the identity of a user; in MRG Messaging, this is done using the SASL framework. Rule-based authorization is a mechanism for specifying the actions that each user is allowed to perform; in MRG Messaging, this is done using an Access Control List (ACL) that is part of the MRG Messaging broker. Encryption is used to ensure that data is not transferred in a plain-text format that could be intercepted and read. Digital signatures provide proof that a given message was sent by a known sender. Encryption and signing are done using SSL (they can also be done using SASL, but SSL provides stronger encryption).

10.1. User Authentication

AMQP uses Simple Authentication and Security Layer (SASL) to authenticate client connections to the broker. SASL is a framework that supports a variety of authentication methods. For secure applications, we suggest **CRAM-MD5**, **DIGEST-MD5**, or **GSSAPI**. The **ANONYMOUS** method is not secure. The **PLAIN** method is secure only when used together with SSL. To use SASL from the python client, you need to install the *python-saslwrapper* rpm (and its dependency *saslwrapper*).

Both the MRG Messaging broker and MRG Messaging clients use the *Cyrus SASL library*[1], a full-featured authentication framework, which offers many configuration options. This section shows how to configure users for authentication with SASL. If you are not using SSL, you should configure SASL to use **CRAM-MD5**, **DIGEST-MD5**, or **GSSAPI** (which provides Kerberos authentication). For information on configuring these and other options in SASL, see the Cyrus SASL documentation at **/usr/share/doc/cyrus-sasl-lib-2.1.22/index.html** for Red Hat Enterprise Linux 5.6.

Important

The **SASL PLAIN** method sends passwords in plain text, and is vulnerable to man-in-the-middle attacks unless SSL (Secure Socket Layer) is also used (see *Section 10.3, "Encryption using SSL"*).

If you are not using SSL, we recommend that you disable **PLAIN** authentication in the broker.

The MRG Messaging broker uses the **auth yes|no** option to determine whether to use SASL authentication. Turn on authentication by setting **auth** to **yes** in **/etc/qpidd.conf**:

```
# /etc/qpidd.conf
#
# Set auth to 'yes' or 'no'

auth=yes
```

10.1.1. Configuring SASL

The SASL configuration file is in **/etc/sasl2/qpidd.conf** for Red Hat Enterprise Linux 5.6.

[1] http://www.cyrusimap.org/

The SASL database contains user names and passwords for SASL. In SASL, a user may be associated with a *realm*. The MRG Messaging broker authenticates users in the **QPID** realm by default, but it can be set to a different realm using the **realm** option:

```
# /etc/qpidd.conf
#
# Set the SASL realm using 'realm='

auth=yes
realm=QPID
```

The SASL database is installed at **/var/lib/qpidd/qpidd.sasldb**; initially, it has one user named **guest** in the **QPID** realm, and the password for this user is **guest**.

Note

The user database is readable only by the `qpidd` user. When run as a daemon, MRG Messaging always runs as the `qpidd` user. If you start the broker from a user other than the `qpidd` user, you will need to either reconfigure SASL or turn authentication off.

Important

The SASL database stores user names and passwords in plain text. If it is compromised so are all of the passwords that it stores. This is the reason that the `qpidd` user is the only user that can read the database. If you modify permissions, be careful not to expose the SASL database.

Add new users to the database by using the **saslpasswd2** command, which specifies a realm and a user ID. A user ID takes the form ***user-id*@*domain*.**.

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

To list the users in the SASL database, use **sasldblistusers2**:

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

If you are using **PLAIN** authentication, users who are in the database can now connect with their user name and password. This is secure only if you are using SSL. If you are using a more secure form of authentication, please consult your SASL documentation for information on configuring the options you need.

10.1.2. Kerberos

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

To use Kerberos, both the MRG Messaging broker and each MRG Messaging 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. 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 MRG Messaging broker, the primary is **qpidd**, the instance is the fully qualified domain name, which you can obtain using **hostname --fqdn**, and the REALM is the Kerberos domain realm. By default, this realm is **QPID**, but a different realm can be specified in qpid.conf, e.g.:

   ```
   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**.

   Note

   When using **GSSAPI** clients you must specify the fully qualified domain name for the broker they are connecting to which corresponds to the principal created for the qpidd service on that host.

   The following script creates a principal for qpidd:

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

   Now create a Kerberos keytab file for the MRG Messaging broker. The MRG Messaging 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, e.g.:

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

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

   ```
   # /etc/qpidd.conf
   ```

```
auth=yes
realm=EXAMPLE.COM
```

Restart the broker to activate these settings.

3. 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
```

Java JMS clients require a few additional steps.

1. 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

2. The client's 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>''
   ```

Please refer to the following documentation for more detail on using Kerberos:

RHEL5
: *Red Hat Enterprise Linux 5: Deployment Guide* [2]

Java
: *Introduction to JAAS and Java GSS-API Tutorials* [3]

[2] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/ch-kerberos.html
[3] http://java.sun.com/j2se/1.5.0/docs/guide/security/jgss/tutorials/index.html

10.2. Authorization

In MRG Messaging, Authorization specifies which actions can be performed by each authenticated user using 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
```

Each line in an ACL file grants or denies specific rights to a user. 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 rajith@QPID all all
acl deny all all
```

On this server, **`rajith@QPID`** can perform any action, but nobody else can. Deny mode is the default, so the previous example is equivalent to the following ACL file:

```
acl allow rajith@QPID all all
```

In deny mode, denying rights to an action is redundant and has no effect.

```
acl allow rajith@QPID all all
acl deny jonathan@QPID all all # This rule is redundant, and has no effect
acl deny all all
```

If the last line in an ACL file is **`acl allow all all`**, ACL uses *allow mode*, and all rights are granted except those that are explicitly denied. The following ACL file allows everyone else to perform any action, but denies **`jonathan@QPID`** all permissions.

```
acl deny jonathan@QPID all all
acl allow all all
```

In allow mode, allowing rights to an action is redundant and has no effect.

```
acl allow rajith@QPID all all # This rule is redundant, and has no effect
acl deny jonathan@QPID all all
acl allow all all
```

Important

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

```
acl allow all all
```

```
acl deny all all
```

Any lines that occur after one of these statements will be ignored:

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

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
```

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
```

10.2.1. 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>]
```

ACL rules can also include a single object name (or the keyword *`all`*) and one or more property name value pairs in the form **`property=value`**

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

Table 10.1. 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 10.2. 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 10.3. 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 10.4. 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.

10.2.2. ACL Syntactic Conventions

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

10.2.3. Specifying ACL Permissions

Now that we have seen the ACL syntax, we will provide representative examples and guidelines for ACL files.

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
```

10.3. Encryption using SSL

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

Enabling SSL for the MRG Messaging 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 client's 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*[4], 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.

--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**literal>, and appending any **DC**literal>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.

Enabling SSL in Clients

C++ clients:

1. 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 10.5. SSL Client Environment Variables for C++ clients

SSL Client Options for C++ clients	
QPID_SSL_USE_EXPORT_POLICY	Use NSS export policy

SSL Client Options for C++ clients	
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.

2. 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.

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

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, e.g.

```
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.

Optimization

This section covers ways to optimize MRG Messaging applications to improve performance. Some optimizations involve the structure of your code, others involve tuning parameters.

Benchmarks can be utilized to determine the expected throughput and latency of MRG Messaging in your environment. Red Hat supplies a set of benchmark applications in the **qpid-cpp-client-devel** package. This package contains:

- **qpid-perftest** for measuring throughput, and
- **qpid-latency-test** for measuring latency.

Each of these programs provide options for testing performance in different conditions. The can be viewed by using the **--help** option at the shell prompt:

```
$ qpid-perftest --help

$ qpid-latency-test --help
```

You can use the benchmarking tools to determine how changing a setting affects performance in your environment, so you can find the best settings to use in your program.

If these benchmarks perform significantly better than your application using the same configuration and settings, it is quite likely that MRG Messaging is not the bottleneck. See "Performance Tips" in *Programming in Apache Qpid* for tips on how to improve performance for applications using the Qpid Messaging API.

Management Tools

There are two kinds of management tools for MRG Messaging.

The MRG Management Console is a web-based tool that can be used to manage brokers, queues and messages within a graphical interface. It is a powerful tool with broad management capabilities. It is described the *MRG Management Console Installation Guide*.

The command-line tools are lightweight management and diagnostic tools designed for use at the shell prompt. They are described in this chapter. Here are the management tools for MRG Messaging that are described in this chapter:

`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 — this tool is described in *Section 7.2, "qpid-cluster"*

`qpid-route`
Configure federated routes among brokers — this tool is described in *Section 8.4, "The qpid-route Utility"*

`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— this tool is described in *Section 7.5.4, "Starting a persistent cluster with no clean store"*

The command line utilities are included in the python client library package. Follow the installation instructions in the *Messaging Installation Guide* and install the **`qpid-tools`** package (this has dependencies on the **`python-qpid`** and **`python-qmf`** packages).

12.1. Using `qpid-config`

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
================================================================================
qmfagent-16a3e0ab-03c0-4a96-a416-7d0bc336d5fd  auto-del excl
qmfagent-d7da3494-c52d-414d-a214-b662158f5edf  auto-del excl
qmfc-v2-hb-rh5_64.6212.1                       auto-del excl --limit-policy=ring
qmfc-v2-rh5_64.6212.1                          auto-del excl
qmfc-v2-ui-rh5_64.6212.1                       auto-del excl --limit-policy=ring
reply-rh5_64.6212.1                            auto-del excl
topic-rh5_64.6212.1                            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
```

Note

For more information on using **qpid-config** to manage queues, see *Chapter 3, Queues*.

6. 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
```

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 a queue, use the **del exchange** command with the name of the exchange to remove:

```
$ qpid-config del exchange exchange_name
```

12.2. 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...
```

12.3. Using qpid-queue-stats

1. View the full list of commands by running the **qpid-queue-stats --help** command from the shell prompt:

```
$ qpid-queue-stats --help
usage: qpid-queue-stats [options]

options:
-h, --help           show this help message and exit
-a BROKER_ADDRESS,   broker-addr is in the form:  [username/password@]
...[output truncated]...
```

2. View the statistics for all queues in the local broker by using the **qpid-queue-stats** alone:

```
$ qpid-queue-stats
Queue Name    Sec Depth Enq Rate    Deq Rate
================================================================================
mgmt-localhost.localdomain.12531       10.00     3     1.40  1.20
mgmt-localhost.localdomain.12531       10.00     3     0.50  0.50
mgmt-localhost.localdomain.12531       10.00     5     0.70  0.50
mgmt-localhost.localdomain.12531       10.00     3     1.50  1.70
mgmt-localhost.localdomain.12531       10.00     2     0.50  0.60
mgmt-localhost.localdomain.12531       10.00     4     0.60  0.40
message_queue          10.00    11      0.37  0.00
mgmt-localhost.localdomain.12531       10.00     2     1.10  1.30
message_queue          10.00     0     0.00  1.10
```

3. To view the statistics for a particular broker, use the **qpid-queue-stats** command and specify the broker. Brokers can be specified in a number of different ways. If the broker requires authentication, specify the username and password separated by a / character and followed by the @ character. The broker itself can be specified by either hostname or IP address, which can be followed by a port number separated by a : character. The format for brokers is:

```
[username/password@] hostname | ip-address [:port]
```

Some valid examples are:

- localhost
- 10.1.1.7:10000
- broker-host:10000
- guest/guest@localhost

More Information

Reporting a Bug

If you have found a bug in MRG Messaging, follow these instructions to enter a bug report:

1. You will need a *Bugzilla*[1] account. You can create one at *Create Bugzilla Account*[2].
2. Once you have a Bugzilla account, log in and click on *Enter A New Bug Report*[3].
3. When submitting a bug report, you will need to identify the product (Red Hat Enterprise MRG), the version (2.0), and whether the bug occurs in the software (component = messaging) or in the documentation (component = Messaging_User_Guide).

Further Reading

- Red Hat Enterprise MRG and MRG Messaging Product Information
  - *http://www.redhat.com/mrg*
- Red Hat Enterprise MRG and MRG Messaging Documentation
  - *http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/*
  - *http://www.redhat.com/mrg/resources/*
- MRG Messaging Users Mailing List
  - Subscribe by sending an email to *`rhemrg-users-list@redhat.com`* with the word *`Subscribe`* in the subject line.

Appendix A. Revision History

Revision 1-0 Thu Jun 23 2011 **Alison Young** *alyoung@redhat.com*

Prepared for publishing

Revision 0.1-8 Fri Jun 03 2011 **Alison Young** *alyoung@redhat.com*

Minor XML updates

Revision 0.1-7 Tue May 31 2011 **Alison Young** *alyoung@redhat.com*

Technical Review minor fix

Revision 0.1-6 Wed May 25 2011 **Alison Young** *alyoung@redhat.com*

Technical Review fixes
BZ#677656 - Replication behaves as a load balancer
BZ#683594 - How to use priority queues

Revision 0.1-5 Fri May 20 2011 **Alison Young** *alyoung@redhat.com*

Technical Review fixes
BZ#617488 - how to resize the message store journal with store resize utilities
BZ#629226 - Chapter 12 updates
BZ#629567 - qpid-cluster-store -c on running clustered brokers marks store clean

Revision 0.1-4 Wed May 18 2011 **Alison Young** *alyoung@redhat.com*

Technical Review fixes
BZ#673225 - clarify meaning of "rec hdr and tail"
BZ#673230 - XID Size
BZ#676572 - --truncate option has not specified default state
BZ#683503 - description of producer flow control update

Revision 0.1-3 Fri May 13 2011 **Alison Young** *alyoung@redhat.com*

BZ#571363 - ACL options added to table 10.4
BZ#585853 - python kerberos client authentication requirements
BZ#683503 - description of producer flow control
BZ#683599 - Using configurable queue threshold alerts updates

Revision 0.1-2 Tue Apr 19 2011 **Alison Young** *alyoung@redhat.com*

BZ#656226 - Qpid federation "queue route"
BZ#683599 - How to use configurable queue threshold alerts

Revision 0.1-1 Tue Apr 05 2011 **Alison Young** *alyoung@redhat.com*

BZ#683586 - Update Further Reading section

Revision 0.1-0 Tue Feb 22 2011 **Alison Young** *alyoung@redhat.com*

Fork from 1.3