

Red Hat Enterprise MRG 1.1

Messaging User Guide

Use, configuration and tuning information for MRG Messaging



Lana Brindley

Red Hat Enterprise MRG 1.1 Messaging User Guide

Use, configuration and tuning information for MRG Messaging

Edition 1

Author

Lana Brindley

lbrindle@redhat.com

Copyright © 2008 Red Hat, Inc

Copyright © 2008 Red Hat, Inc. This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version of the OPL is presently available at <http://www.opencontent.org/openpub/>).

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein 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
PO Box 13588 Research Triangle Park, NC 27709 USA

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 Messaging Tutorial.

| | |
|---|-----------|
| Preface | v |
| 1. Document Conventions | v |
| 1.1. Typographic Conventions | v |
| 1.2. Pull-quote Conventions | vii |
| 1.3. Notes and Warnings | viii |
| 2. We Need Feedback! | viii |
| 1. MRG Messaging Concepts | 1 |
| 1.1. The basis of MRG Messaging | 1 |
| 1.2. How MRG Messaging operates | 1 |
| 1.3. Exchange Types | 2 |
| 2. Management Tools | 7 |
| 2.1. MRG Management Console | 7 |
| 2.2. Command Line Tools | 8 |
| 2.2.1. Using qpuid-config | 8 |
| 2.2.2. Using qpuid-tool | 11 |
| 2.2.3. Using qpuid-queue-stats | 14 |
| 3. Queues | 17 |
| 4. Sessions | 23 |
| 5. Transactions | 25 |
| 6. Persistence | 27 |
| 6.1. Persistent Queues | 28 |
| 6.1.1. Estimating Resources | 28 |
| 7. Clustering and federation | 33 |
| 7.1. Messaging Clusters | 33 |
| 7.2. Federation | 36 |
| 8. Authentication | 43 |
| 8.1. User Authentication | 43 |
| 8.2. Authorization | 44 |
| 8.3. Encryption using SSL | 48 |
| 9. Optimization | 53 |
| 10. More Information | 57 |
| A. Revision History | 59 |

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. *Messaging* — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *Realtime* — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *Grid* — 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, but includes persistence options, additional components, Linux kernel optimizations, and operating system services not found in the Qpid implementation. We have worked closely with companies that rely heavily on high performance messaging, and created a system to meet their real-world needs.

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 *Messaging Tutorial*.

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

¹ <http://cwiki.apache.org/qpid/>

² <https://fedorahosted.org/liberation-fonts/>

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps 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.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

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

Press **Enter** to execute the command.

Press **Ctrl-Alt-F1** to switch to the first virtual terminal. Press **Ctrl-Alt-F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, 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; dialogue box text; labelled 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.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

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:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules (MPMs)*. Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types 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 are presented and highlighted 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

A Note is a tip or shortcut or alternative approach 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 Important boxes won't cause data loss but may cause irritation and frustration.



Warning

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

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/bugzilla/> 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.

MRG Messaging Concepts

1.1. The basis of MRG Messaging

Advanced Message Queuing Protocol

AMQP is an open-source messaging protocol. It offers increased flexibility and interoperability across languages, operating systems, and platforms. AMQP is the first open standard for high performance enterprise messaging. More information about AMQP is available at the [AMQP website](http://www.amqp.org/)¹. The full protocol specification is also [available for download](http://www.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1)².

Apache Qpid

Qpid is an Apache Incubator project that implements the AMQP protocol. It is a multi-platform messaging implementation that delivers transaction management, queuing, distribution, security and management. Development on MRG Messaging is also contributed back upstream to the Qpid project. More information can be found on the [Apache Qpid website](http://cwiki.apache.org/qpid/)³.

Red Hat Enterprise MRG Messaging

MRG Messaging is an open source messaging distribution that uses the AMQP protocol. MRG Messaging is based on Qpid, but includes persistence options, additional components, Linux kernel optimizations, and operating system services not found in the Qpid implementation.

1.2. How MRG Messaging operates

MRG Messaging was designed to provide a way to build distributed applications in which programs exchange data by sending and receiving messages. A message can contain any kind of data. Middleware messaging systems allow a single application to be distributed over a network and throughout an organization without being restrained by differing operating systems, languages, or network protocols. Sending and receiving messages is simple, and MRG Messaging provides guaranteed delivery and extremely good performance.

In MRG Messaging a *message producer* is any program that sends messages. The program that receives the message is referred to as a *message consumer*. If a program both sends and receives messages it is both a message producer and a message consumer.

The *message broker* is the hub for message distribution. It receives messages from message producers and uses information stored in the message's headers to decide where to send it on to. The broker will normally attempt to send a message until it gets notification from a consumer that the message has been received.

Within the broker are *exchanges* and *queues*. Message producers send messages to exchanges, message consumers subscribe to queues and receive messages from them.

The message headers contain *routing* information. The *routing key* is a string of text that the exchange uses to determine which queues to deliver the message to. *Message properties* can also be defined for settings such as message durability.

¹ <http://www.amqp.org/>

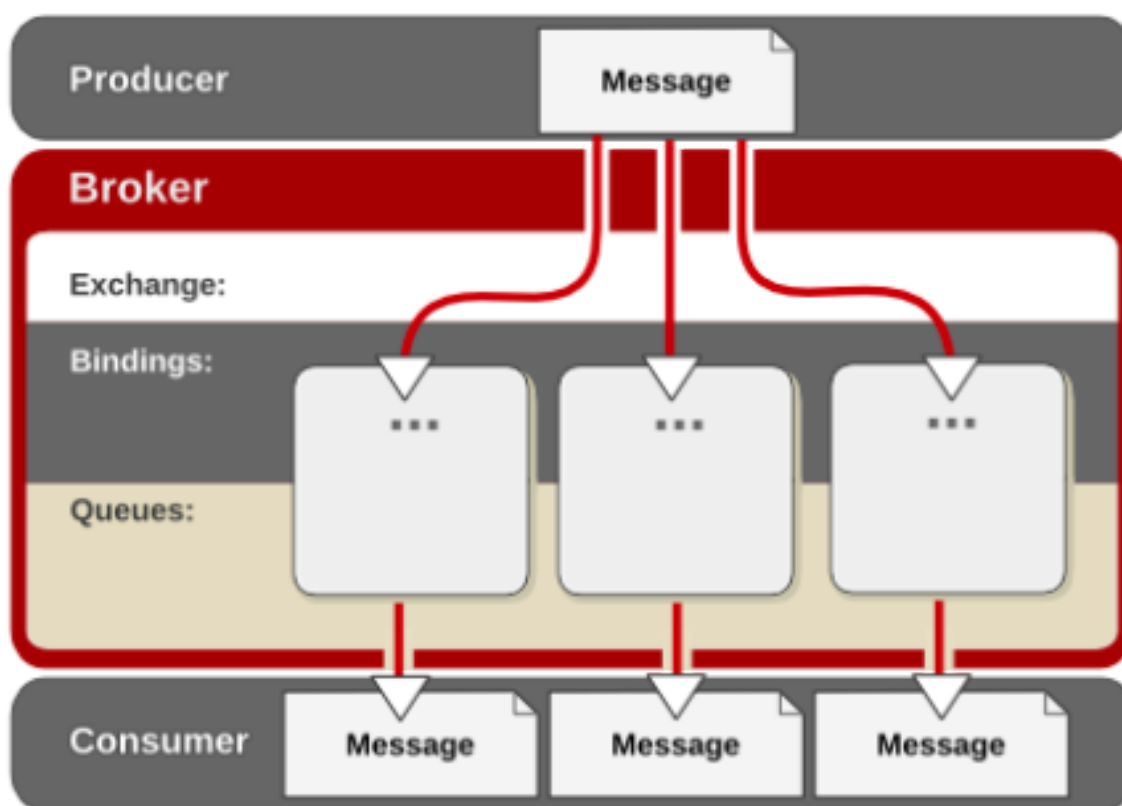
² <http://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1>

³ <http://cwiki.apache.org/qpid/>

A *binding* defines the relationship between an exchange and a message queue. A queue must be bound to an exchange in order to receive messages from it. When an exchange receives a message from a message producer, it examines its active bindings, and routes the message to the corresponding queue. Consumers read messages from the queues to which they are subscribed. Once a message is read, it is removed from the queue and discarded.

In the following diagram, a producer sends messages to an exchange. The exchange reads the active bindings and places the message in the appropriate queue. Consumers then retrieve messages from the queues.

Producer Consumer

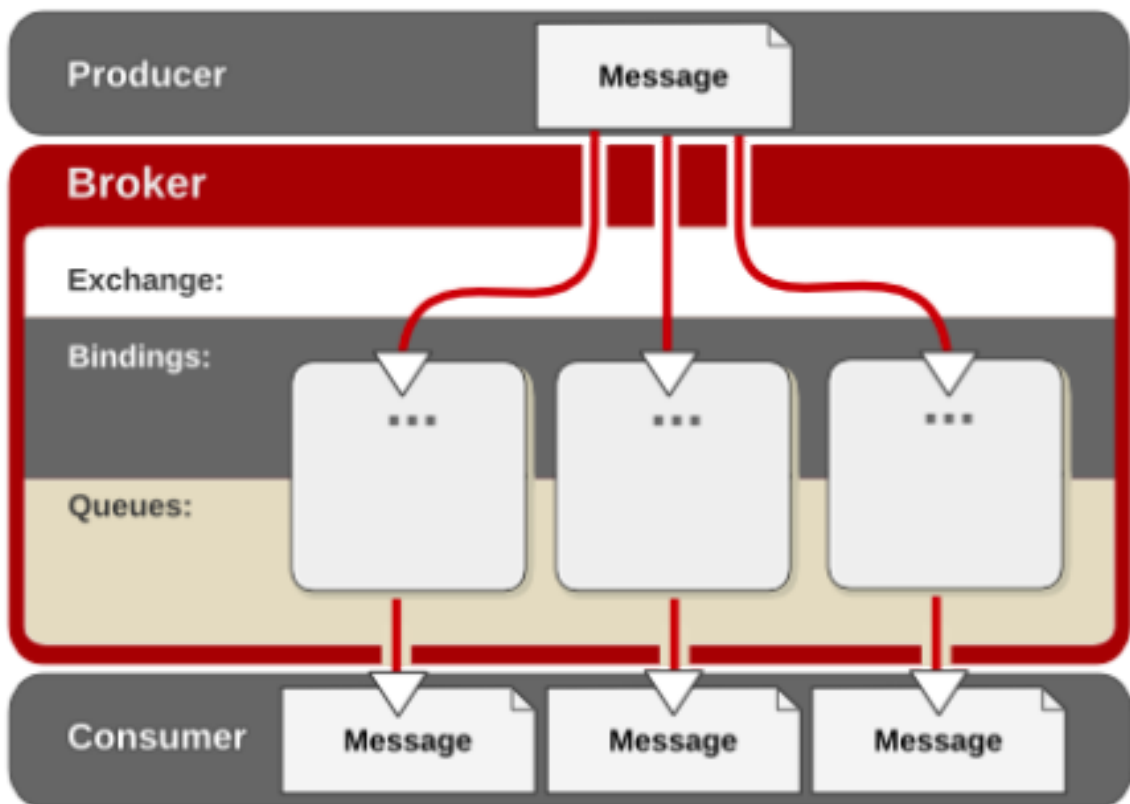


1.3. Exchange Types

Every message is sent through an exchange, which determines how to distribute the message. In MRG Messaging there are three standard exchanges and a custom exchange type:

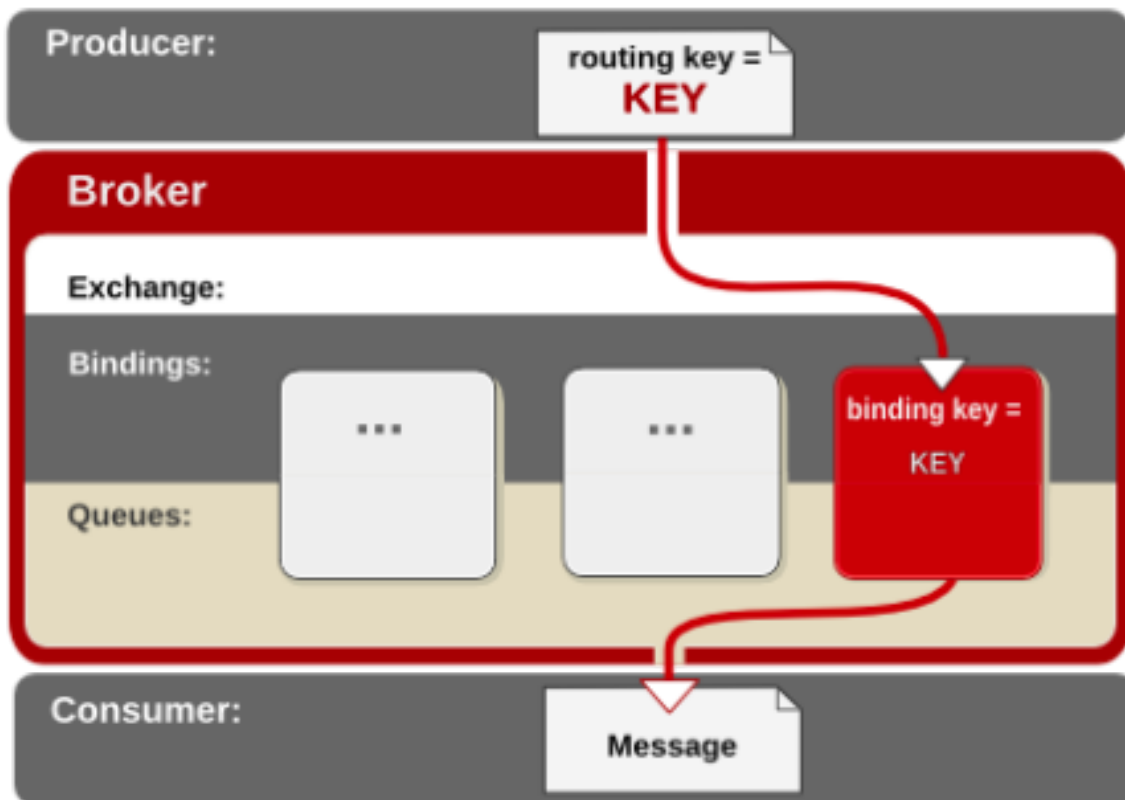
- Fanout
- Direct
- Topic
- Custom Exchange

Fanout Exchange



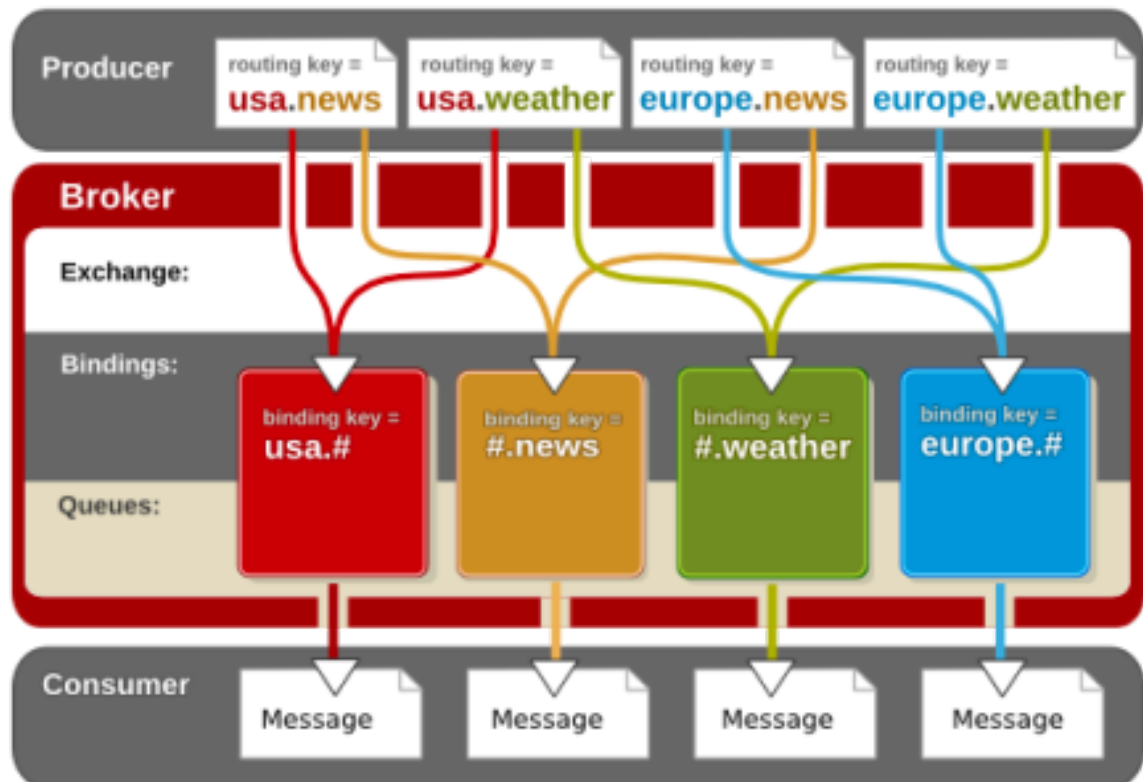
The *fanout exchange* will distribute messages to every queue. Any routing information provided by the producer is ignored.

Direct Exchange



In a *direct exchange* the broker will search for matches between the message's routing key and the queues' binding keys. If the broker finds an exact match, it will deliver the message to that queue.

Topic Exchange



A topic exchange uses multi-part keys to find matches between routing keys and binding keys. Topic exchanges can also use wildcard characters to find matches.

This example demonstrates the use of multi-part keys in a topic exchange

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.#*

So, 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*

Example 1.1. Using multi-part keys with a topic exchange

Custom Exchange Types

Exchange types that do not fit the standard set of exchanges are referred to as *custom exchanges*. MRG Messaging provides a custom exchange that operates with the Extensible Markup Language XML.

The custom XML exchange sends messages written in XML. The messages contain bindings written in the XML query language XQuery. It is implemented as an optional module in MRG Messaging.

Management Tools

There are a variety of management tools available 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. The command-line tools are lightweight management and diagnostic tools designed for use at the shell prompt.

2.1. MRG Management Console

If you already have the Red Hat Enterprise MRG yum repository installed on your system, running the `yum groupinstall "MRG Management"` command will install the correct packages for the MRG Management Console. For comprehensive installation information, see the *MRG Management Console Installation Guide*.

Register New Brokers

To begin using the MRG Management Console you will need to register some brokers.

1. Select **Register New Brokers** from the main window.

MRG Management - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost/index.html?frame=main.brokersadd.main.m=brokersadd

Messaging Systems

Main

Actions: 0 pending, 0 completed, 0 failed

Register New Brokers

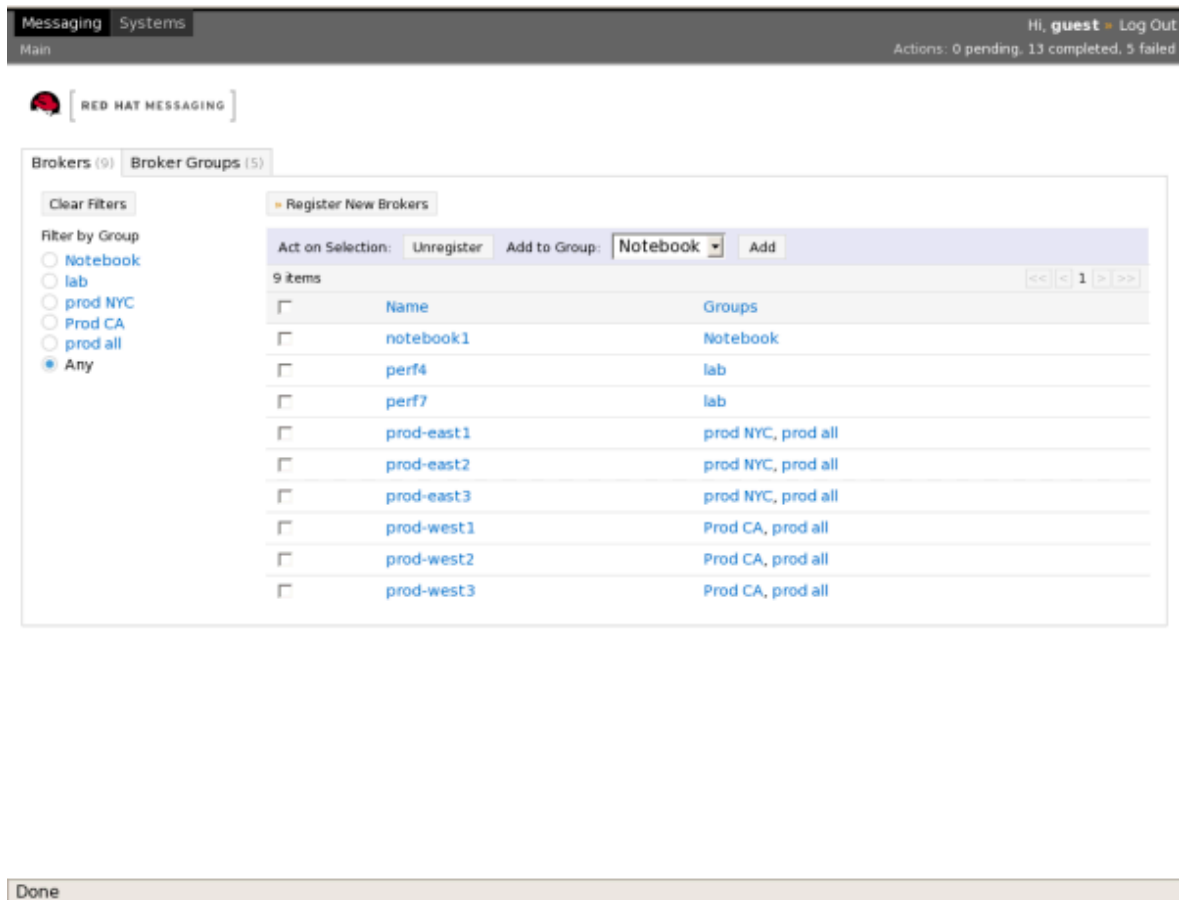
| Name | Host Name or IP Address | Initial Group |
|----------------------|-------------------------|---------------|
| <input type="text"/> | <input type="text"/> | None |
| <input type="text"/> | <input type="text"/> | None |
| <input type="text"/> | <input type="text"/> | None |

More Entries

Help Cancel Submit

Done

2. Enter the name of the new broker and its host name or IP address. Click on **Submit** to save your information and return to the Brokers screen. You should be able to see your new broker listed.
3. From the main screen you can select the name of your new broker to edit its properties, or select brokers to perform actions.



2.2. Command Line Tools

MRG Messaging contains a number of command line utilities for monitoring and configuring messaging brokers. <http://cwiki.apache.org/qpid/mgmtc.html>

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

perftest

Measures throughput on a variety of scenarios, using adjustable parameters

The command line utilities are included in the python client library package. Follow the installation instructions in the *Messaging Installation Guide* and install the **python-qpid** and **amqp** packages.

2.2.1. 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
```

| Store | | Size | | | | |
|---------|---------|------|----------|---------------|--------|---|
| Durable | AutoDel | Excl | Bindings | (files x file | pages) | Queue Name |
| N | N | N | 1 | | | pub_start |
| N | | N | 1 | | | pub_done |
| N | | N | 1 | | | sub_ready |
| N | | N | 1 | | | sub_done |
| N | | N | 1 | | | perftest0 |
| N | Y | N | 2 | | | mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15 |
| N | Y | N | 2 | | | repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 |
| N | Y | N | 2 | | | mgmt-df06c7a6-4ce7-426a-9f66-da91a2a6a837 |
| N | Y | N | 2 | | | repl-df06c7a6-4ce7-426a-9f66-da91a2a6a837 |

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
```

2.2.2. Using **qp**id-**tool**

1. The **qp**id-**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 **qp**id-**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 **qp**id: 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. **qp**id-**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: list
Management Object Types:
ObjectType      Active  Deleted
=====
qpid.binding    21      0
qpid.broker     1        0
qpid.client     1        0
qpid.exchange   6        0
qpid.queue      13       0
qpid.session    4        0
qpid.system     1        0
qpid.vhost      1        0
```

4. You can choose which objects to list by also specifying a class:

```

qpid: list qpid.system
Objects of type qpid.system
ID    Created    Destroyed    Index
=====
1000  21:00:02    -           host

```

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 Description | Type | Unit | Access | Notes |
|--|--------------|---------|------------|-------|
| 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)

```

- To exit the tool and return to the shell, type **quit** at the prompt:

```

qpidd: quit
Exiting...

```

2.2.3. Using qpidd-queue-stats

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

```

$ qpidd-queue-stats --help
usage: qpidd-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]...

```

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

```

$ qpidd-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 statistic for a particular broker, use the **qpidd-view-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

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 1, MRG Messaging Concepts](#).

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.

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 **qpidd-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 cancelled. It will not be deleted until at least one session has subscribed and then cancelled 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 behaviour 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.

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 messages 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.

This example demonstrates the use of LVQ.

```
{
#include "qpid/client/QueueOptions.h"

QueueOptions qo;
qo.setOrdering(LVQ);

session.queueDeclare(arg::queue=queue, arg::arguments=qo);

.....
string key;
qo.getLVQKey(key);

....
```

For each message, set the into application headers before transfer

```
message.getHeaders().setString(key, "RHT");
}
```

Example 3.1. Setting the LVQ

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.

This example demonstrates the use of **Persist Last Node**

```
#include "qpid/client/QueueOptions.h"

QueueOptions qo;
qo.clearPersistLastNode();

session.queueDeclare(arg::queue=queue, arg::durable=true,
rg::arguments=qo);
```

Example 3.2. Using **Persist Last Node**

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 2.2.1, "Using qpid-config"](#).

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

```
$ qpid-config queues
```

| Durable | AutoDel | Excl | Bindings | Store Size (files x file pages) | Queue Name |
|---------|---------|------|----------|------------------------------------|------------|
| N | N | N | 1 | | pub_start |
| N | | N | 1 | | pub_done |
| N | | N | 1 | | sub_ready |
| 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 create queues using AMQPs **queue declare** command. This command allows the durable, exclusive, auto-delete and alternate-exchange properties to be specified. Any **qpidd** specific options can be passed in the arguments field. See the *MRG Messaging API documentation* for the client language you wish to use for more details.

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 the broker

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 acknowledgement, 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.

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 queue. *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

A persistence library allows MRG Messaging to store messages and queue configuration, ready to be reloaded in the event of machine or network failure. When the persistent store module is loaded, it allows messages and other persistent state information to be recovered when a broker is restarted.

When a transaction occurs, a journal called the Transaction Prepared List (or TPL) is initialized. This journal stores the state of the transaction in the broker as it occurs. If the broker then crashes or is stopped for any reason, the journal can restore the information to the broker when it is restarted. In this way, any in-progress transactions that were either prepared or committed when the broker failed will be recovered.

For storage of the broker itself, and for exchange and binding information, the [Berkeley Database](#)¹ is used to store and recover information.

When adjusting parameters for persistence, there are three settings specific to the TPL. This allows alternate settings for those messages that are recovered.

In order for messages to be stored the persistence store must be loaded. The **--store-dir** command specifies the directory used for the persistence store and any configuration information. The default directory is **/var/lib/qpid**. See [Table 6.1, "Persistence Options"](#) for options on how to change this behavior.



Important

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



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.*

In addition to loading the persistence store, queues and messages also need to be identified as durable. This can be done in the client application or by using the **qpid-config** command line tool. See the *MRG Messaging Tutorial* for more information about creating client applications.

It is important to set the store journal size to match the anticipated persistent message queue size. The store uses a fixed-size circular file buffer, so it is possible for the store to run out of space to queue messages if consumers are slow or messages are very large.

Each queue that is marked persistent will cause the broker to create an instance of the store together with its files. The broker will stop accepting persistent messages when approximately 80% of the capacity is reached. Consuming of messages (dequeuing), however will be allowed to continue. Once the dequeued messages have cleared sufficient space, message queuing will continue as normal. As a rule of thumb, the journal capacity should be about double the size expected to be stored on the disk at any one moment in time.

¹ <http://www.oracle.com/technology/products/berkeley-db/index.html>



Warning

A totally full condition on the journal (in which there is no more write space in the circular buffer) is a fatal condition. It is possible to read the messages in a full journal, but not to dequeue them, as dequeuing requires the ability to write dequeue records. To prevent this, message queuing is disabled at 80% capacity.

| Persistence Options | |
|---|---|
| --store-dir <i>DIRECTORY</i> | Specifies the directory used for storage of persistence configuration information. The default is <code>/var/lib/qpidd</code> . |
| --num-jfiles <i>NUMBER</i> | Set the number of files for each instance of the persistence journal. The default is 8. |
| --jfile-size-pgs <i>NUMBER</i> | Set the size of each journal file in multiples of 64KB. The default is 24. |
| --wcache-page-size <i>NUMBER</i> | 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 <i>NUMBER</i> | Set the number of files for each instance of the TPL journal. The default is 8. |
| --tpl-jfile-size-pgs <i>NUMBER</i> | Set the size of each TPL journal file in multiples of 64KB. The default is 24. |
| --tpl-wcache-page-size <i>NUMBER</i> | 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. |

Table 6.1. Persistence Options

6.1. Persistent Queues

When a broker creates persistent queues, the store module is used to create, write and maintain the queue journal files on disk for as long as the queue exists. However, opening a number of persistent queues at the same time can cause system resource limitations. This section discusses some considerations to avoid reaching default limits or exhausting system resources.

6.1.1. Estimating Resources

Estimate the number of simultaneous persistent queues

Determine the number of simultaneous persistent queues that will be required. This number is the primary value that determines resource consumption, so it will be helpful to estimate typical, best and worst case scenarios.

Estimate the required number of file handles

Once the likely range of simultaneous persistent queues is known, it is possible to estimate the number of file handles that will be consumed. This calculation is effected by the broker store settings.

The broker store parameter **--num-jfiles** (or the equivalent setting in the broker configuration file) sets the default number of files used for all persistent queues that will be created on that broker – and thus determines the rate at which file handles will be consumed. This parameter defaults to 8 files.



Note

The MRG Management Console may be used to override these defaults and create individual queues that have file geometry parameters (including the number of files) that differ from this setting.

The store uses a single journal instance for storing transaction boundaries. This journal is called the *Transaction Prepared List* (TPL). The TPL is initialized when the first transaction occurs on any persistent queue, and has its own geometry parameters, separate from the other journal instances. The **--tpl-num-jfiles** parameter (or the equivalent setting in the broker configuration file) has little effect on overall file handle consumption and can be ignored in the calculation.

The broker opens one write file handle for each file. It will also open one for the read pipeline on initialization. Other file handles are opened temporarily during recovery, but are closed again. Once normal operations begin, the read pipeline will eventually be invalidated by overwrite, and the read handle closed and released. However, if flow-to-disk is initiated on a queue the read pipeline may be reinitialized, and a read file handle will be opened for that queue until it is invalidated by overwrite once again.



Note

When using default settings, expect to consume nine file handles per queue with no transactions.

This example uses the formulas given above to calculate the required number of file handles.

Using the default settings, 6,144 simultaneous persistent queues requires $6,144 * 9 = 55,296$ file handles.

Example 6.1. Calculating the number of file handles

Estimate the required number of AIO event handles

The store reserves one AIO (Asynchronous Input/Output) event handle for each page of the journal memory cache for both read and write pipelines. Since the overall size of each of these caches is fixed at 1MB, the size of the pages will directly affect the number of pages and hence the number of AIO event handles being reserved on queue creation. The broker parameter **--wcache-page-size** sets the journal write cache page size. The total number of pages is obtained by dividing the total cache size by the page size. The journal read cache page size is not adjustable because it is used internally only to keep the cache full for read operations.

The write cache page size affects message storage latency for persistent queues and messages. Smaller page sizes means that messages are written to disk more quickly, although this occurs at the

expense of throughput and a greater number of pages and consumed AIO event handles. Conversely, larger page sizes improve overall persistent message throughput and lowers the page count, at the expense of message latency.

The kernel manages this resource on a system-wide basis. If any other processes use the AIO system, then their AIO event handle usage must be added to any estimates.



Note

When using default settings, expect to consume 64 AIO event handles per queue with no transactions.

The number of pages in the cache is obtained by dividing the page size by the fixed total cache size.

This example uses the formula given above to calculate the required number of AIO event handles.

Using the default settings, 6144 simultaneous queues requires $6,144 * 64 = 393,216$ AIO handles.

Example 6.2. Calculating the number of AIO event handles

Estimate the required memory

There are two main memory allocations resulting from persistent queue declarations.

Journal cache: each persistent queue is allocated 1 MB for a journal write cache and 1 MB for a journal read cache. The journal cache sizes cannot be adjusted, however the number of pages used for the journal write cache can be adjusted.

Kernel: the kernel reserves space for expected events. This amount is then rounded up to the memory page size (4KB). An overhead of 312 bytes for every AIO context is then added to the total.



Note

When using default settings, expect to consume about 2MB per queue. The kernel effects are negligible.

This example uses the formulas given above to estimate the required memory.

Using the default settings, 5,000 simultaneous persistent queues will require about 10GB of cache memory. If the average queue depth is 1,000 messages and the average message size is 1KB, the broker will need $5,000 * 1,000 * 1 = 5,000,000\text{KB}$ (5GB) to keep the messages in the queues.

Example 6.3. Estimating the required memory

Changing the resource limits

1. Change the file handle limit by switching to the root user and opening the `/etc/security/limits.conf` in your preferred text editor. Add the following line for the `qpidd` user to set both the hard and soft limits:

```
qpidd - nofile 32768
```

2. Change the AIO handle limit by opening the `/etc/sysctl.conf` and adding the following line:

```
fs.aio-max-nr = 262144
```

3. Check that the system that will run the broker has sufficient memory to support the required number of queues.
4. After making the changes, perform a reboot of the system to start using the new settings. Check the appropriate files to ensure the changes have persisted before starting the broker.

Common errors

There are two errors that can occur as a result of resource problems.

```
jexception 0x0400 fcntl::clean_file() threw JERR_FCNTL_OPENWR: Unable to  
open file for write.  
(open() failed: errno=24 (Too many open files))
```

This error occurs if the broker runs out of available file handles. Under typical default conditions, the broker store will consume all available handles at around 110 queues.

```
jexception 0x0103 pmgr::initialize() threw JERR__AIO: AIO error.  
(io_queue_init() failed: errno=11 (Resource temporarily unavailable))
```

This error occurs if the broker runs out of available AIO event handles.

Clustering and federation

7.1. Messaging Clusters

A *Messaging Cluster* is a group of brokers that act as a single broker. Changes on any broker are replicated to all other brokers in the same Messaging Cluster, so if one broker fails, its clients can fail-over to another broker without loss of state. The brokers in a Messaging Cluster may run on the same host or on different hosts.

Messaging Clusters are implemented using OpenAIS, which provides a reliable multicast protocol, tools, and infrastructure for implementing replicated services. You must install and configure OpenAIS to use MRG broker groups.

Messaging Clusters can be used together with Red Hat Clustering Services (RHCS) by starting brokers with the `--cluster-cman` option

Message Clusters in MRG Messaging uses an Active/Active model. In this model, all possible brokers are active at all times. Producers and consumers can be connected to any broker in the cluster. Additionally, any broker can be killed and restarted, and the cluster will retain its operational state. This model provides scalability and enhanced load-balancing.

Clustering requirements

1. The user that runs the broker must be a member of the group **root** and must have their primary group set to **ais**. A user called 'qpidd' with the correct membership is created during installation.

Starting the AIS service

Clustering requires the Application Interface Specification (AIS) to be running. AIS requires the **openais** package.

1. Ensure the **openais** service is not running:

```
# service openais stop
Stopping OpenAIS daemon (aisexec):          [ OK ]
```

2. Create a file at `/etc/ais/openais.conf` using the following information:

```
totem {
  version: 2
  secauth: off
  threads: 0
  interface {
    ringnumber: 0
    bindnetaddr: 192.168.1.0    # Modify for your network
    mcastaddr: 226.94.1.1
    mcastport: 5405
  }
}
```

```
logging {
    to_syslog: yes
}

amf {
    mode: disabled
}
```

3. The **bindnetaddr** entry should be set to the subnet you will use for cluster multicast. Use the same subnet for all hosts in the cluster. You can find the subnet with `/sbin/ifconfig`. For example:

```
# /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1E:37:88:72:8A
          inet addr:192.168.1.103  Bcast:192.168.1.255
          Mask:255.255.255.0
```

In this case the local address is 192.168.1.103 and the subnet mask is 255.255.255.0. The subnet is the bitwise AND of the mask and the local address, 192.168.1.255.

4. Restart the openais service:

```
# service openais start
Starting OpenAIS daemon (aisexec):          [ OK ]
```

Setting up the redundant ring

OpenAIS supports a *redundant ring protocol (RRP)*, which uses two physically separate networks for cluster communication, so that the cluster can continue to operate if one network fails.

1. Choose the replication mode for your environment. RRP has 3 modes:

- active

Active replication can offer slightly lower latency in faulty network environments, however it can reduce throughput

- passive

Passive replication can nearly double the speed from transmit to delivery, but also carries the potential for the protocol to become bound to a single CPU

- none

Disables redundant ring.

2. To enable RRP make the following changes to **openais.conf**:

- a. In the totem section, add **rrp_mode=active** or **rrp_mode=passive**

- b. Add a second interface section with a different **bindnetaddr** for your second network.

For more information about configuring openAIS see the `openais.conf(5)` man page.



Note

Make sure you run brokers that are members of a cluster as a user with primary group "ais".

3. Once the cluster is running, you can view the active state from **qpidd-tool**:

```
Object of type org.apache.qpid.cluster:cluster: (last sample time:
15:07:23)
```

```
Type      Element      110
```

```
=====
property  brokerRef      102
property  clusterName    test_cluster
property  clusterID      1a911c67-6fcd-4ffa-8fc8-0dc74ec14ec0
property  publishedURL   amqp:tcp:10.16.18.96:5672
property  clusterSize    2
property  status        ACTIVE
property  members      amqp:tcp:10.16.18.96:5672
amqp:tcp:10.16.18.96:5673
```

You can also see this information from the MRG Management Console. See [Section 2.1, "MRG Management Console"](#) for more information.

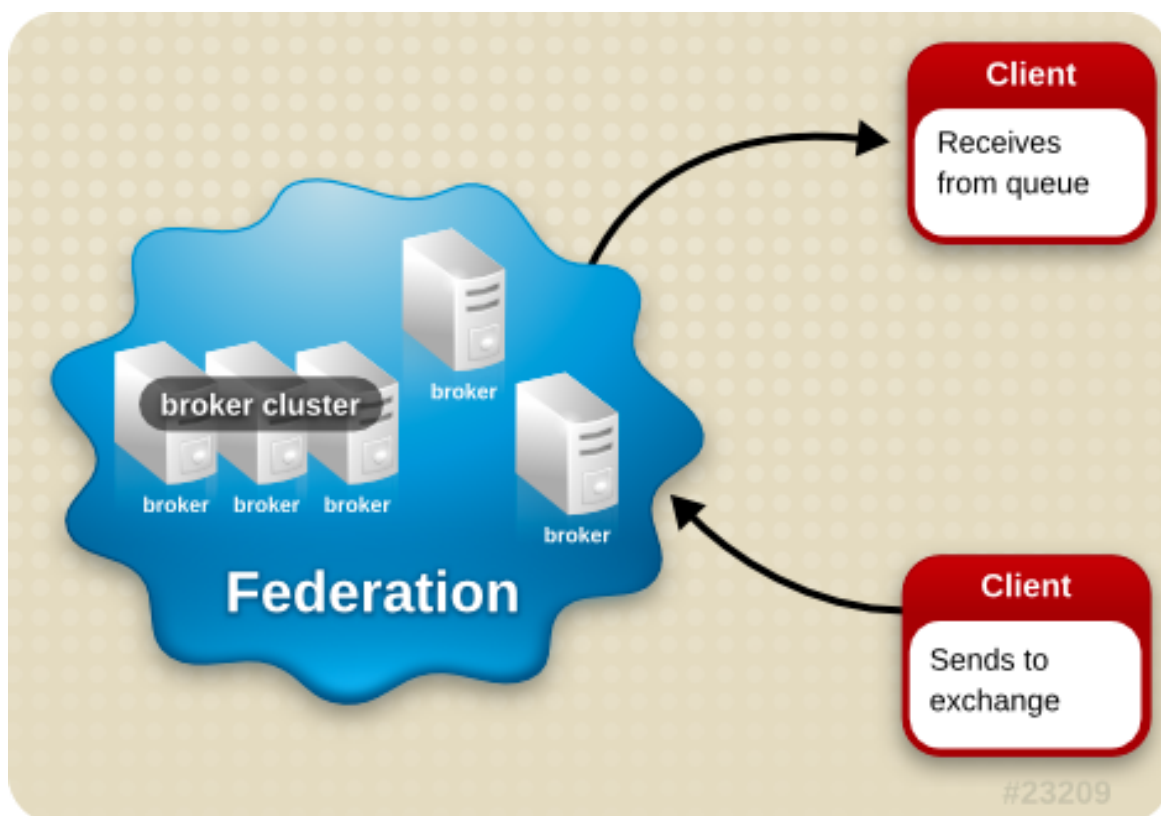
| Options for clustering | |
|----------------------------|--|
| --cluster-name NAME | Name of the Messaging Cluster to join. A Messaging Cluster consists of all brokers started with the same cluster-name and openais configuration. |
| --cluster-url URL | <p>An AMQP URL containing the local address advertised to clients for fail-over connections. This is different for each host. By default, all local addresses are advertized. You only need to set this if</p> <ol style="list-style-type: none"> 1. Your host has more than one active network interface. 2. You want to restrict client fail-over to a specific interface or interfaces. <p>The URL is of the form amqp:tcp:<host>:<port>[, tcp:<host>:<port> ...] For example: amqp:tcp:192.168.1.103:5672, tcp:192.168.1.105:</p> |

| Options for clustering | |
|------------------------|--|
| --cluster-cman | <p>CMAN protects against the "split-brain" condition, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. When "split-brain" occurs, each of the sub-clusters can access shared resources without knowledge of the other sub-cluster, resulting in corrupted cluster integrity.</p> <p>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 documentation¹ for more detail.</p> <p>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.</p> |

Table 7.1. Options for clustering

7.2. Federation

Federation is used to provide geographical distribution of brokers. A number of individual brokers, or clusters of brokers, can be federated together. This allows client machines to see and interact with the federation as though it were a single broker. Federation can also be used where client machines need to remain on a local network, even though their messages have to be routed out.



Federation is used primarily for connecting disparate locations across a wide area network. Full connectivity across an enterprise can be achieved while keeping local message traffic isolated to a single location. Departmental brokers can be specified with individual policies that control inter-departmental message traffic flow.

Some applications can benefit from having a broker co-resident with the client. This is good for situations where the client produces data that must be delivered reliably but connectivity can not be guaranteed. In this case, a co-resident broker provides queueing and durability that is not available in the client on its own.

Federation bridges disjointed IP networks. Message brokers can be configured to allow message connectivity between networks where there is no IP connectivity. For example, an isolated, private IP network can have messaging connectivity to brokers in other outside IP networks.

Links and routes

Federation is configured through a series of links and routes.

A *link* is a connection between two brokers that allows messages to be passed between them. A link is a transport level connection (using a protocol such as TCP, RDMA, or SSL) that is initiated by a broker and accepted by another broker. The broker that initiates the link is considered the client in the connection. The broker that receives that connection will not treat it any differently from any other client connection, other than annotating it as being for federation.

Routes are the paths that messages take from one broker to another, and can run along one or more links to the final destination. A route is associated with an AMQP session established over the link connection. A route controls the flow of messages across the link between brokers, and multiple routes can share the same link. Messages will flow over a single route in only one direction. For bi-directional connectivity a pair of routes must be created, one for each direction of message flow.

Routes always consist of a session and a subscription for consuming messages. Depending on the configuration, a route can have a private queue on the source broker with a binding to an exchange on that broker.

Routes can be configured from either the source broker or the destination broker. In most cases, they are configured from the source broker. Sometimes it is more convenient to configure links and routes from the destination broker. For instance, if brokers are co-resident with data sources, each source can be configured to send data to the central broker.

Making connections

When a link is created on a broker, it will immediately attempt to establish a transport-level connection to another broker. If the connection fails due to a communication error, it will continue trying. The retry interval begins at 2 seconds and, as more attempts are made, grows out to 64 seconds. It will continue to try to make the connection every 64 seconds until explicitly told to stop trying. If the connection fails due to an authentication problem, it will not continue to retry.

Durable links and routes

Links can be made durable by using the **--durable** option when creating it in the **qpidd-route** tool. Durable links will persist between broker restarts. When the broker starts again the link will be restored and will begin establishing connectivity.

Routes can also be made durable with the **--durable** option, as long as they run over durable links.

Dynamic routing

Dynamic routing creates a set of distributed exchanges. All the brokers in a network collectively behave in the same way as a single exchange in a single broker. Each client connects to its local broker, where it can bind queues and publish messages to the distributed exchange.

When configuring dynamic routing, it is only necessary to define which pairs of brokers are connected by a unidirectional route. Queue configuration and bindings are then handled automatically by the brokers in the network.

When a consuming client binds a queue to the distributed exchange, information about that binding is sent to the other brokers in the network. This ensures that any messages matching the binding will be forwarded to the client's local broker. Messages published to the distributed exchange are forwarded to other brokers only if there are remote consumers to receive the messages. The dynamic binding protocol ensures that messages are routed only to brokers with eligible consumers. This includes topologies where messages must make multiple hops to reach the consumer.

Exchange routes

An *exchange route* is like a dynamic route except that the exchange binding is statically set when it is created, and does not dynamically track changes in the network. When an exchange route is created, a private auto-delete, exclusive queue is created on the source broker. The queue is bound to the exchange with the specified key and the destination broker subscribes to the queue with the specified exchange. Only one exchange name is supplied, as exchange routes require that the source and destination exchanges have the same name.

Queue routes

A *queue route* is created when the destination broker subscribes to a pre-existing queue on the source broker. The queue does not need to be any particular exchange. Queue routes can be used to connect exchanges of different names and types. They can also be used to distribute or balance traffic across multiple destination brokers.

Using **qpidd-route** to manage federation

The **qpidd-route** utility provides the ability to configure and manage links and routes. If a route is created and a link does not already exist, **qpidd-route** will automatically create that link.



Note

The command line utilities are included in the python client library package. Follow the installation instructions in the *Messaging Installation Guide* to install the **python-qpidd** and **amqp** packages.

1. Commands are entered at the shell prompt in the format

```
# qpidd-route [options] [command] [destination broker] [source broker]
```

When adding and deleting routes exchange and routing key information is also required.

2. 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

3. To add links, use the **link add** command, specifying both the destination broker and the source broker:

```
$ qpidd-route -v link add destination-broker localhost
```

4. To delete links, use the **link delete** command, specifying both the destination broker and the source broker of the link you wish to remove:

```
$ qpidd-route -v link del destination-broker localhost
```

5. To view a complete list of links to a particular broker, use the **link list** command, specifying broker you wish to view:

```
$ qpid-route link list localhost:10001
```

| Host | Port | Transport | Durable | State | Last Error |
|-----------|-------|-----------|---------|-------------|--------------------|
| localhost | 10002 | tcp | N | Operational | |
| localhost | 10003 | tcp | N | Operational | |
| localhost | 10009 | tcp | N | Waiting | Connection refused |

6. Create dynamic routes using the **dynamic** option. When creating a dynamic routing network, the type and name of the exchange must be the same on each broker.



Note

Unless you intend all messaging to be federated, it is strongly recommended that dynamic routes are not created using standard exchanges.

Firstly, create an exchange on each broker:

```
$ qpid-config -a localhost:10003 add exchange topic fed.topic
$ qpid-config -a localhost:10004 add exchange topic fed.topic
```

Now, create the dynamic routes:

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

7. To add static exchange routes, use the **route add** command. Specify both the destination broker and the source broker. It also requires an exchange type (direct, fanout or topic) and a routing key:

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

Use # characters as wildcards when specifying the routing key for a topic exchange, to send to all possible matches.

8. To view a complete list of the routes to a particular broker, use the **route list** command, specifying the broker you wish to view:

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


9. To add static queue routes, firstly create a queue on the source broker:

```
$ qpid-config -a localhost:10002 add queue public
```

Now, create the queue route to the new queue:

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

10. A more comprehensive view of a network can be achieved using the **route map** command, with a single broker name. **qpid-route** will then attempt to recursively find all of the brokers related to the starting broker, and map the relationships it finds:

```
$ 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
```

11. To remove all messages currently en-route to a particular broker, use the **route list** command, specifying the broker you wish to flush:

```
$ qpid-route -v route flush localhost
```

| Options for using qpid-route to Manage Federation | |
|---|---|
| -v | Verbose output. |
| -q | Quiet output, will not print duplicate warnings. |
| -d | Make the configuration change durable. |
| -e | Delete link after deleting the last route on the link |

Table 7.2. Options for using **qpid-route** to manage federation

Authentication

In MRG Messaging, authentication is provided by a Simple Authentication and Security Layer (SASL) and authorization is managed by an Access Control List (ACL). Mozilla's Network Security Services Library (SSL) provides encryption for secure password management.

8.1. User Authentication

MRG Messaging uses Simple Authentication and Security Layer (SASL) for identifying and authorizing incoming connections to the broker, as mandated in the AMQP specification. SASL provides a variety of authentication methods. While MRG Messaging clients primarily implement the **PLAIN** method, the broker uses the [Cyrus SASL library](http://cyrusimap.web.cmu.edu/)¹ to allow for a full SASL implementation.



Important

The **PLAIN** authentication method sends passwords in cleartext. For complete security, it is advised that SSL (Secure Socket Layer) is also used. See [Section 8.3, "Encryption using SSL"](#)

Enabling and Using SASL Plain Authentication

To use the default SASL PLAIN authentication mechanism implemented by the MRG Messaging client libraries, either use the default username and password of *guest*, which are included in the database at `/var/lib/qpidd/qpidd.sasl` on installation, or add your own accounts.

1. Add new users to the database by using the **saslpasswd2** command. The User ID for authentication and ACL authorization uses the form **user-id@domain..**

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

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

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

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



Note

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

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

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

¹ <http://cyrusimap.web.cmu.edu/>

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

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

```
auth=no  
  
auth=yes
```

4. The SASL configuration file is in `/etc/sasl2/qpidd.conf` for Red Hat Enterprise Linux 5 and `/usr/lib/sasl2/qpidd.conf` for Red Hat Enterprise Linux 4.

For information on using a different configuration, use your web browser to view the Cyrus SASL documentation at `/usr/share/doc/cyrus-sasl-lib-2.1.22/index.html` for Red Hat Enterprise Linux 5 or `/usr/share/doc/cyrus-sasl-2.1.19/index.html` for Red Hat Enterprise Linux 4.

8.2. Authorization

Authorisation in MRG Messaging is achieved through the use of an Access Control List (ACL). This is a list that specifies which users are authorized to access the system.

Using ACL

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

```
$ qpidd --help  
...[output truncated]...  
ACL Options:  
--acl-file FILE (policy.acl) The policy file to load from, loaded from  
data dir
```

2. To start using the ACL, you will need to specify the file to use. This is done by using the `--acl-file` command with a path and filename. The filename should have a `.acl` extension:

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

You can now view the file with the `cat` command and edit it in your preferred text editor. If the path and filename is not found, `qpidd` will fail to start.

3. These permissions can be used when creating the ACL file:

```
allow  
    Allow rule
```

allow-log

Allow rule and log the action in the event log

deny

Deny rule

deny-log

Deny rule and log the action in the event log

4. The following actions are valid:

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

5. The following object types are supported:

queue

A queue

exchange

An exchange

broker

The broker

link

A federation or inter-broker link

method

Management or agent or broker method

6. Wild cards can be used on properties that are a string. The following properties are supported:

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

When editing the ACL file, the following rules 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.
- 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**

This example demonstrates correctly formatted ACL entries.

Specifying 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
```

Specifying rules:

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

Always include the last, default rule:

```
acl deny all all
```

Do not allow *guest* to access and log some QMF management methods:

```
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
```

Example 8.1. An example ACL file

8.3. Encryption using SSL

Encryption and certificate management for **qpidd** is provided by Mozilla's Network Security Services Library (SSL), through the **ssl.co** module. This module is installed by default in MRG Messaging.

Enabling SSL for **qpidd**

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, the client's certificate will also need to be signed by a CA and trusted by 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 will need to specify the location of the database containing the certificate and key to use. This can be done using the **ssl-cert-db** option.

The database is created and managed by the Mozilla Network Security Services (NSS) **certutil** tool. More information can be found on the [Mozilla website](#)², including tutorials on setting up and testing SSL connections.



Note

The certificate database will generally be password protected. The password can be specified at the command prompt when starting **qpidd**. Alternatively, create a file containing the password and direct **qpidd** at the file location using the **ssl-cert-password-file** option.

2. Load the broker module:

```
$ qpidd --load-module /libs/ssl.so
```

3. The available SSL options are:

--ssl-use-export-policy

Use NSS export policy

ssl-cert-password-file PATH

File containing password to use for accessing certificate database

--ssl-cert-db PATH

Path to directory containing certificate database

--ssl-cert-name NAME

Name of the certificate to use

--ssl-port NAME

Port on which to listen for SSL connections

ssl-require-client-authentication

Forces clients to authenticate in order to establish an SSL connection

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

Enabling SSL

C++ clients:

1. SSL is provided through the **sslconnector.so** module. This module is installed and loaded by default in MRG Messaging. To enable the module, you will need to specify the

location of the database containing the certificate and key to use. This can be done using the **ssl-cert-db** option in **/etc/qpid/qpidc.conf** or using the environment variable **QPID_SSL_CERT_DB**.

2. If **ssl-require-client-authentication** is active on **qpidd**, the clients certificate will also need to be verified. To do this, use **--ssl-cert-name** and, if necessary, **--ssl-cert-password-file**.
3. To open an SSL enabled connection, the application will need to specify *ssl* as the value for the protocol setting in the **ConnectionSettings** instance passed to **Connection::open()**.

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 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 client:

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

.Net clients:

1. If the broker requires authentication, you will need a certificate signed by a CA, trusted by your client.
2. Use the **connectSSL** method instead of the usual connect method for the client interface. The string for the **connectSSL** signature is:

```
public void connectSSL(String host, int port,
String virtualHost, String username, String password,
String serverName, String certPath, bool rejectUntrusted)
```

For these values:

- *host*: Host name on which a Qpid broker is deployed
- *port*: Qpid broker port
- *virtualHost*: Qpid virtual host name
- *username*: Username

- *password*: Password
- *serverName*: Name of the SSL server
- *certPath*: Path to the X509 certificate to be used when the broker requires client authentication
- *rejectUntrusted*: When true the connection will not be established if the broker is not trusted - the server certificate must be added in your truststore.

| SSL Client Options for C++ clients | |
|---|--|
| --ssl-use-export-policy | Use NSS export policy |
| --ssl-cert-password-file <i>PATH</i> | File containing password to use for accessing certificate database |
| --ssl-cert-db <i>PATH</i> | Path to directory containing certificate database |
| --ssl-cert-name <i>NAME</i> | Name of the certificate to use |

Table 8.1. SSL Client Options for C++ clients

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 **qpidd-perftest** package. This package contains:

- **perftest** for measuring throughput, and
- **latencytest** for measuring latency.

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

```
$ perftest --help
$ latencytest --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.



Note

This section is still under development. If you have found any tricks for optimizing your installation, why not let us know, and get it included in this document? Just follow the instructions in [Reporting a Bug](#).

Try these first

When you start optimizing your installation of MRG Messaging, try these things first:

1. Use asynchronous communication with the broker where possible. The Qpid APIs allow both synchronous and asynchronous communication. When performing a large amount of message transfers, asynchronous communication is much faster.

From C++

The **Session** interface will issue commands synchronously. The **AsyncSession** interface will issue them asynchronously. You can convert from one to the other easily using **async(session)** and **sync(session)**. You can also synchronise at a specific point using **session.sync()**

From Python

session.auto_sync = False will turn off the default synchronous behaviour. You can synchronise explicitly using **session.sync(timeout)**

From Java JMS

All messaging is asynchronous in the Java JMS client, however persistent messaging offers both synchronous and asynchronous publishing. Synchronous messaging publishing is more reliable, but can be quite slow.

Synchronous publishing can be set by adding the **-Dsync_persistence=true** option as a global property. This will make all publishing slower.

Synchronous publishing can also be set as a connection option. More information on doing this is available from the *MRG Messaging Tutorial*

2. Accept messages in batches rather than one by one. This decreases network traffic, while still guaranteeing delivery.

From C++

If you are using automatic acceptance, set the **autoAck** setting in **SubscriptionSettings** to a value greater than 1 (the value is the size of the batch that will be accepted). If managing accepts manually, do so in batches. For example:

```
//a batch of messages is identified by a SequenceSet containing
//the relevant Ids:
qpId::framing::SequenceSet batch;

//add message ids to the set:
batch.add(message.getId());

//etc
subscription.accept(batch);

//or
session.messageAccept(batch);
```

From Python

```
batch = RangedSet() ...

#add message ids to the set
batch.add(message.id)

#etc
session.message_accept(batch)
```

From Java JMS

Check the *MRG Messaging Tutorial* for instructions.

For exclusive, auto-delete queues there is often no real value to using the explicit accept mode. Turning off the need to accept messages at all may also offer a performance gain. In C++, this is achieved by specifying **ACCEPT_MODE_NONE** as the **acceptMode** in **SubscriptionSettings**. In Python you would specify the accept mode when

issuing a subscription request: `session.messageSubscribe(queue='q', accept_mode=session.accept_mode.none)`. In JMS the queues used for topic subscriptions will do this automatically.

3. Use pre-fetch. Pre-fetching instructs the broker to deliver messages to the client in anticipation of them being consumed.

From C++

Pre-fetch is controlled through the **flowControl** setting on **SubscriptionSettings**. The default is an unlimited pre-fetch which may overwhelm the client. For example, to set a prefetch of 100 messages:

```
SubscriptionSettings settings;
settings.flowControl = FlowControl::messageWindow(100);
```

From Python

The default is an unlimited pre-fetch. For example, to reduce that to a prefetch window of 10 messages:

```
session.message_subscribe(destination="my-subscriber", queue="my-queue")
session.message_set_flow_mode(destination="my-subscriber", session.flow_mode.window)
session.message_flow(destination="my-subscriber", session.credit_unit.message, 100)
session.message_flow(destination="my-subscriber", session.credit_unit.byte, 0xFFFFFFFF)
```

From Java JMS

Use either the **DUPS_OK** or **AUTO_ACK** acknowledgement mode. For compliance with the JMS specification, the **AUTO_ACK** acknowledgement mode should always be used with a pre-fetch value of 0. This ensures one message is received and acknowledged at a time, which results in slow performance.

The maximum pre-fetch amount can be set by adding the **-Dsync_prefetch=800** option as a global property. The default value is 1000.

The maximum pre-fetch amount can also be set as a connection option. More information on doing this is available from the *MRG Messaging Tutorial*

4. Consider enabling **TCP - NODELAY**. This will generally improve latency but can also impact throughput. However if you are using very small transactions with a synchronous commit, this option can also improve throughput.

From C++

Set the **tcpNodeLay** option to *true* on the **ConnectionSettings** instance passed to **Connection::open()**.

```
#include <qpid/client/ConnectionSettings.h>

ConnectionSettings connectionSettings;
```

```
connectionSettings.host = "localhost";
connectionSettings.port = 5672;
connectionSettings.tcpNoDelay = true;

connection.open(connectionSettings);
```

5. Consider tuning the maximum frame size used. This will affect the degree to which the broker tries to batch messages for delivery to clients. To improve latency, try reducing the value from the default 64kb. This will not prevent messages larger than the max frame size being sent, but it will impact the maximum size of the message headers. Picking a value that is large enough for the majority of messages to fit in a single content frame is likely to be most optimal.

From C++

Set the **maxFrameSize** option on the **ConnectionSettings** instance passed to **Connection::open()**.

```
#include <qpid/client/ConnectionSettings.h>

ConnectionSettings connectionSettings;

connectionSettings.host = "localhost";
connectionSettings.port = 5672;
connectionSettings.maxFrameSize = 65535;

connection.open(connectionSettings);
```

From Java JMS

This option is not configurable under Java JMS

6. Consider using bounds to control the size of the outgoing message queue. This specifies the maximum number of buffers that the outgoing message queue can hold.

From C++

Set the **bounds** property on the **ConnectionSettings** instance passed to **Connection::open()**.

```
#include <qpid/client/ConnectionSettings.h>

ConnectionSettings connectionSettings;

connectionSettings.host = "localhost";
connectionSettings.port = 5672;
connectionSettings.maxFrameSize = 65535;
connectionSettings.bounds = 4;

connection.open(connectionSettings);
```

7. Experiment with different options using the **perfctest** tool, available from the **qpiddc-perfctest** package.

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 (1.1), 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://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 1.1 Thu Apr 2 2009 Lana Brindley lbrindle@redhat.com
BZ #491173 - corrected error in SASL instructions

Revision 1.0 Thu Feb 12 2009 Lana Brindley lbrindle@redhat.com
BZ #478501 - Estimating Resources for large numbers of persistent queues

Revision 0.14 Wed Jan 21 2009 Lana Brindley lbrindle@redhat.com
BZ #480568 - Durable Queues

Revision 0.13 Mon Jan 19 2009 Lana Brindley lbrindle@redhat.com
Added links to product page

Revision 0.12 Tue Jan 13 2009 Lana Brindley lbrindle@redhat.com
BZ#452123 - PersistLastNode

Revision 0.11 Mon Jan 12 2009 Lana Brindley lbrindle@redhat.com
BZ#479423 - LVQ
BZ#477282 - Deleting Queues

Revision 0.10 Tue Nov 25 2008 Jonathan robie
jonathan.robie@redhat.com
Reworking of the cluster section.

Revision 0.9 Thu Nov 20 2008 Lana Brindley lbrindle@redhat.com
Minor updates prior to releasing document to Quality Engineering

Revision 0.8 Tue Nov 18 2008 Lana Brindley lbrindle@redhat.com
Minor updates to Clustering chapter arising from technical review
Minor updates to Optimization chapter arising from technical review
Minor updates to Concepts chapter arising from technical review

Revision 0.7 Mon Nov 17 2008 Lana Brindley lbrindle@redhat.com
Updated Clustering Information
Updated Optimization Information
Minor updates to Concepts chapter arising from technical review

Revision 0.6 Fri Nov 14 2008 Lana Brindley lbrindle@redhat.com
Updated Security Information - BZ #470378

Appendix A. Revision History

Minor updates to Concepts chapter arising from technical review

Revision 0.5 Fri Nov 14 2008 Lana Brindley lbrindle@redhat.com
Adding federation and clustering information arising from the technical review

Revision 0.4 Thu Nov 13 2008 Lana Brindley lbrindle@redhat.com
Changes arising from technical review

Revision 0.3 Tue Nov 4 2008 Lana Brindley lbrindle@redhat.com
Sessions - Bugzilla #465385
Queues - Bugzilla #465384
Transactions
Optimization/Tuning
Ethernet cards config - BZ#457922

Revision 0.2 Fri Oct 24 2008 Lana Brindley lbrindle@redhat.com
Updated Persistence chapter - Bugzilla #456498
Updated Federation chapter
Updated Security chapter
Updated Management Tools chapter

Revision 0.1 Wed Aug 6 2008 Lana Brindley lbrindle@redhat.com
Initial Document Creation