

# Red Hat Enterprise MRG 1.1

## Messaging Tutorial

AMQP Programming Tutorial for C++, Java, Python, and C#



Jonathan Robie

# **Red Hat Enterprise MRG 1.1 Messaging Tutorial**

## **AMQP Programming Tutorial for C++, Java, Python, and C#**

### **Edition 1.1**

Author Jonathan Robie  
Editor Lana Brindley  
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 shows you how to write programs for the MRG Messaging component of the Red Hat Enterprise MRG distributed computing platform using the Apache Qpid API. It also gives basic information on downloading and installing MRG Messaging. For more complete information on how to download and install MRG Messaging see the MRG Messaging Installation Guide.

<b>Preface</b>	<b>vii</b>
1. Document Conventions .....	vii
1.1. Typographic Conventions .....	viii
1.2. Pull-quote Conventions .....	ix
1.3. Notes and Warnings .....	x
2. We Need Feedback! .....	x
<b>1. Initial Concepts</b>	<b>1</b>
1.1. Fanout Exchange .....	2
1.2. Direct Exchange .....	3
1.3. Topic Exchange .....	4
1.4. Custom Exchange Types .....	5
<b>2. Examples Overview</b>	<b>7</b>
<b>3. Installing MRG Messaging</b>	<b>9</b>
3.1. Installing MRG Messaging on Red Hat Enterprise Linux 5 .....	9
3.2. Installing MRG Messaging on Red Hat Enterprise Linux 4 .....	10
3.3. Starting the Broker .....	10
<b>4. Using MRG Messaging with Python</b>	<b>13</b>
4.1. Creating and Closing Sessions .....	13
4.2. Writing Direct Applications in Python. ....	14
4.2.1. Running the Direct Examples .....	14
4.2.2. Declaring and Binding a Queue .....	15
4.2.3. Publishing Messages to a Direct Exchange .....	16
4.2.4. Reading Messages from the Queue .....	16
4.2.5. Reading Messages from a Queue using a Listener .....	17
4.3. Writing Fanout Applications in Python .....	18
4.3.1. Running the Fanout Examples .....	18
4.3.2. Consuming from a Fanout Exchange .....	19
4.3.3. Publishing Messages to the Fanout Exchange .....	20
4.4. Writing Publish/Subscribe Applications in Python .....	20
4.4.1. Running the Publish-Subscribe Examples .....	20
4.4.2. The Topic Publisher .....	22
4.4.3. The Topic Subscriber .....	23
4.5. Writing Request/Response Applications in Python .....	25
4.5.1. Running the Request/Response Examples .....	25
4.5.2. The Server Application .....	26
4.5.3. The Client Application .....	28
4.6. XML-based Routing in Python .....	29
4.6.1. Running the XML-based Routing Examples .....	29
4.6.2. Declaring an XML Exchange, Declaring and Binding a Queue .....	30
4.6.3. Publishing to an XML Exchange .....	31
4.6.4. Reading from the Message Queue .....	32
4.7. Durable Queues and Durable Messages in Python .....	32
4.8. Using Transactions in Python .....	33
4.9. Logging in Python client applications .....	33
<b>5. Using MRG Messaging with C++</b>	<b>35</b>
5.1. Creating and Closing Sessions .....	35
5.2. Writing Direct Applications in C++ .....	37
5.2.1. Running the Direct Examples .....	37
5.2.2. Declaring and Binding a Queue .....	38

5.2.3. Publishing Messages to a Direct Exchange .....	38
5.2.4. Reading Messages from the Queue .....	39
5.3. Writing Fanout Applications in C++ .....	40
5.3.1. Running the Fanout Examples .....	40
5.3.2. Consuming from a Fanout Exchange .....	42
5.3.3. Publishing Messages to the Fanout Exchange .....	43
5.4. Writing Publish/Subscribe Applications in C++ .....	43
5.4.1. Running the Publish-Subscribe Examples .....	44
5.4.2. Publishing Messages to a Topic Exchange .....	45
5.4.3. Reading Messages from the Queue .....	46
5.5. Writing Request/Response Applications in C++ .....	49
5.5.1. Running the Request/Response Examples .....	49
5.5.2. The Client Application .....	50
5.5.3. The Server Application .....	52
5.6. XML-based Routing in C++ .....	53
5.6.1. Running the XML-based Routing Examples .....	54
5.6.2. Declaring an XML Exchange, Declaring and Binding a Queue .....	55
5.6.3. Publishing to an XML Exchange .....	55
5.6.4. Reading from the Message Queue .....	56
5.7. Durable Queues and Durable Messages in C++ .....	57
5.8. Using Transactions in C++ .....	58
5.9. Optimizing message transfer with asynchronous sessions in C++ .....	58
5.10. Handling Failover in C++ Connections .....	59
5.10.1. Sending Messages in a <b>FailoverManager::Command</b> .....	60
5.10.2. Receiving Messages with a <b>FailoverManager::Command</b> .....	61
5.10.3. Choosing Brokers for Reconnect .....	62
5.11. Using logging in C++ .....	62
<b>6. Using MRG Messaging with Java JMS</b> .....	<b>65</b>
6.1. Java JMS Client Compatibility and Interoperability .....	65
6.2. Creating and Closing Connections and Sessions with JNDI .....	66
6.2.1. Basic JNDI Programming for MRG Messaging .....	66
6.2.2. JNDI Properties for MRG Messaging .....	67
6.2.3. Connection URLs .....	68
6.2.4. Binding URLs .....	68
6.3. Creating and Closing Connections and Sessions with AMQP .....	69
6.4. Writing Direct Applications in Java JMS .....	70
6.4.1. Running the Direct Examples .....	70
6.4.2. JNDI Properties .....	72
6.4.3. Publishing Messages to a Queue .....	72
6.4.4. Reading Messages from the Queue with a Message Consumer .....	73
6.4.5. Reading Messages from the Queue using a Message Listener .....	75
6.5. Writing Fanout Applications in Java JMS .....	77
6.5.1. Running the Fanout Examples .....	77
6.5.2. JNDI Properties .....	78
6.5.3. Reading Messages from a Queue with a Message Consumer .....	79
6.5.4. Reading Messages from the Queue using a Message Listener .....	80
6.5.5. Publishing Messages to a Fanout Exchange .....	83
6.6. Writing Publish/Subscribe Applications in Java JMS .....	84
6.6.1. Running the Publish/Subscribe Examples .....	84
6.6.2. JNDI Properties .....	86
6.6.3. Publishing Messages to a Topic .....	86

---

6.6.4. Reading Messages from the Queue .....	88
6.7. Writing Request/Response Applications in Java JMS .....	90
6.7.1. JNDI Properties .....	91
6.7.2. Running the Request/Response Examples .....	91
6.7.3. Client .....	92
6.7.4. The Server .....	94
6.8. Durability and Persistence in Java JMS .....	95
6.9. Using Transactions in Java JMS .....	96
6.10. Logging in Java clients .....	96
<b>7. Using MRG Messaging with .NET .....</b>	<b>99</b>
7.1. Creating and Closing Sessions .....	99
7.2. Writing Direct Applications in .NET .....	100
7.2.1. Running the Direct Examples .....	100
7.2.2. Reading Messages from the Queue .....	102
7.2.3. Publishing Messages to a Direct Exchange .....	103
7.3. Writing Fanout Applications .....	104
7.3.1. Running the Fanout Examples .....	104
7.3.2. Consuming from a Fanout Exchange .....	105
7.3.3. Publishing Messages to the Fanout Exchange .....	107
7.3.4. Writing Publish/Subscribe Applications .....	107
7.3.5. Running the Publish-Subscribe Examples .....	107
<b>A. Revision History .....</b>	<b>115</b>

---

---

# Preface

This tutorial will teach you how to write MRG Messaging applications in C++, Python, Java (using the JMS API), and C# (for .NET). To run the programs in this tutorial, you will need to download and install MRG Messaging and be able to start the broker and run a sample application. These steps are described in [Chapter 3, Installing MRG Messaging](#), and described in more depth in the *MRG Messaging Installation Guide*.

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](#)<sup>1</sup>, but includes persistence, 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.

- MRG Messaging is flexible. It easily supports most common messaging paradigms, including store-and-forward, distributed transactions, publish-subscribe, content-based routing, and market data distribution.
- MRG Messaging is interoperable. It implements the Advanced Message Queuing Protocol (AMQP), which is a free and open standard for messaging.
- MRG Messaging supports clients written in many languages, including Java (JMS), C++, Python, and C# (for .Net). Perl, and Ruby clients will be available soon.
- MRG Messaging supports many platforms, including Linux, Windows, and Unix.
- MRG Messaging is fast. MRG Messaging delivers the highest performance and reliability available.
- MRG Messaging is designed for Linux. The C++ broker (which is fully compatible with the Java broker) can integrate directly with the Linux kernel. MRG Messaging is optimized to take full advantage of the Linux kernel, and track Linux kernel developments that might be leveraged for further optimization. And the C++ broker can be directly integrated with the cluster executive as a native cluster service.
- MRG Messaging is reliable, providing guaranteed delivery of messages.
- MRG Messaging is based on proven technology. AMQP is already being used in production systems, where it is serving very high message volumes; for example, one bank has a worldwide deployment that delivers over 100 million messages per day in a 7 hour trading window.
- MRG Messaging supports advanced features including multiple direct-write, persistence, and integration with operating system clustering facilities.
- MRG Messaging is open source. You can see the code, change it, and learn from it.
- MRG Messaging can be used as a standard Linux service. It can be used to support features like virtualization, security, grid computing, and distributed operating system services.

## 1. Document Conventions

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

---

<sup>1</sup> <http://cwiki.apache.org/qpid/>

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)<sup>2</sup> set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

### 1.1. Typographic Conventions

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

#### Mono-spaced Bold

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

---

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



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\_Tutorial*

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.



# Initial Concepts

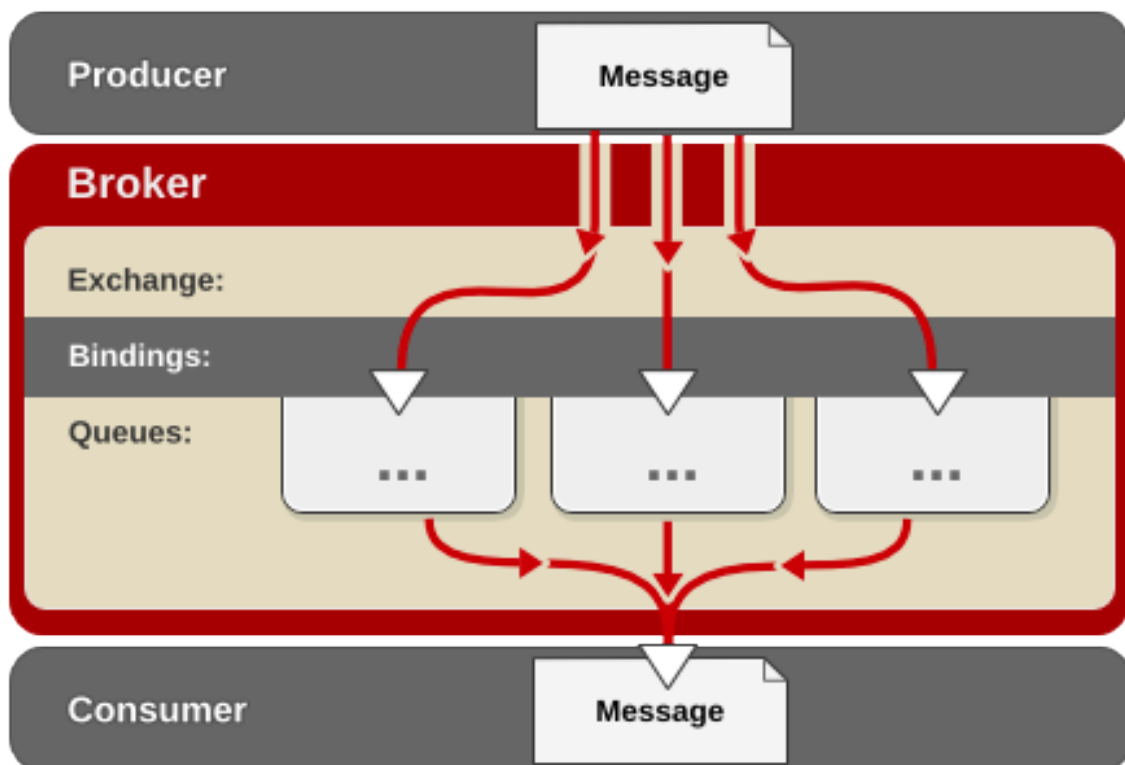
## The AMQP Model

MRG Messaging implements the [AMQP specification](http://www.amqp.org)<sup>1</sup>, which was written to create an open standard for interoperable messaging. AMQP defines both a wire level protocol (the transport layer) and higher level semantics for messaging (the functional layer). It is completely free to use and is being developed by the AMQP Working Group. AMQP is currently in draft and will be submitted to a standards body once it is completed.

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

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

## Producer Consumer



The AMQP Model: Message producers write messages to exchanges, message consumers read messages from queues

<sup>1</sup> <http://www.amqp.org>

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

An *exchange* accepts messages from message producers and routes them to message queues if the message meets the criteria expressed in a *binding*. A binding defines the relationship between an exchange and a message queue, specifying which messages should be routed to a given queue. For instance, a binding might state that all messages with a given routing key should be sent to a particular queue. If a queue is not bound to an exchange, it does not receive any messages from that exchange.

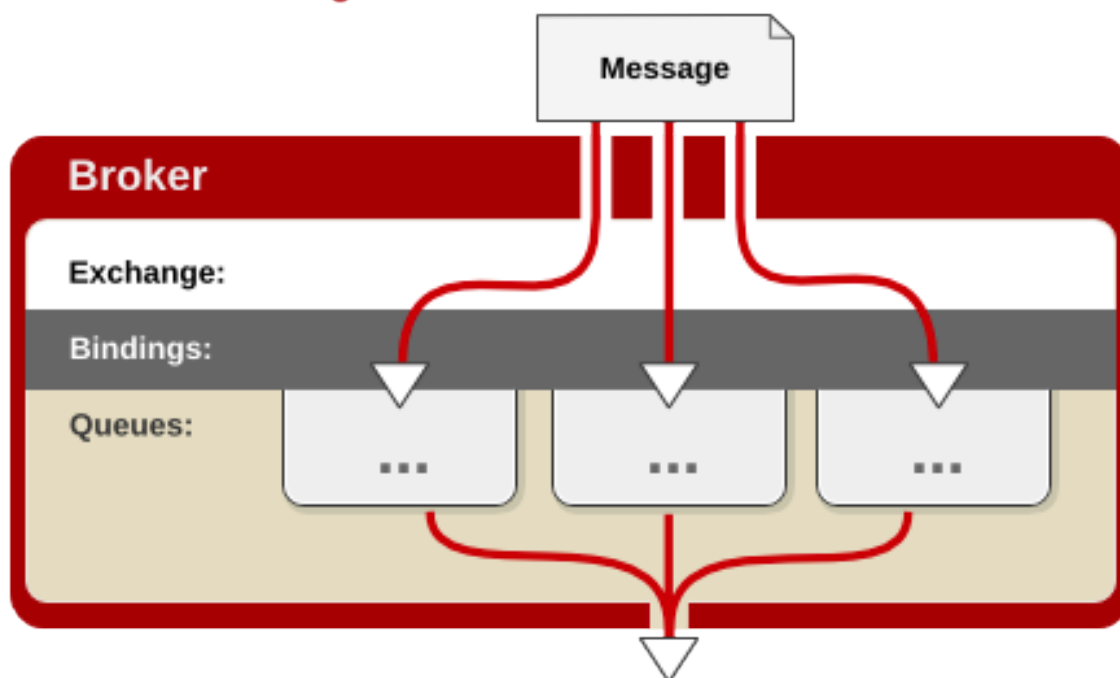
A *message queue* holds messages and delivers them to the *message consumers* that subscribe to the queue. A message consumer can create, subscribe to, share, use, or destroy message queues (as long as they have permission to do so). A message queue may be *durable*, which means that the queue is never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

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

### 1.1. Fanout Exchange

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

## Fanout Exchange

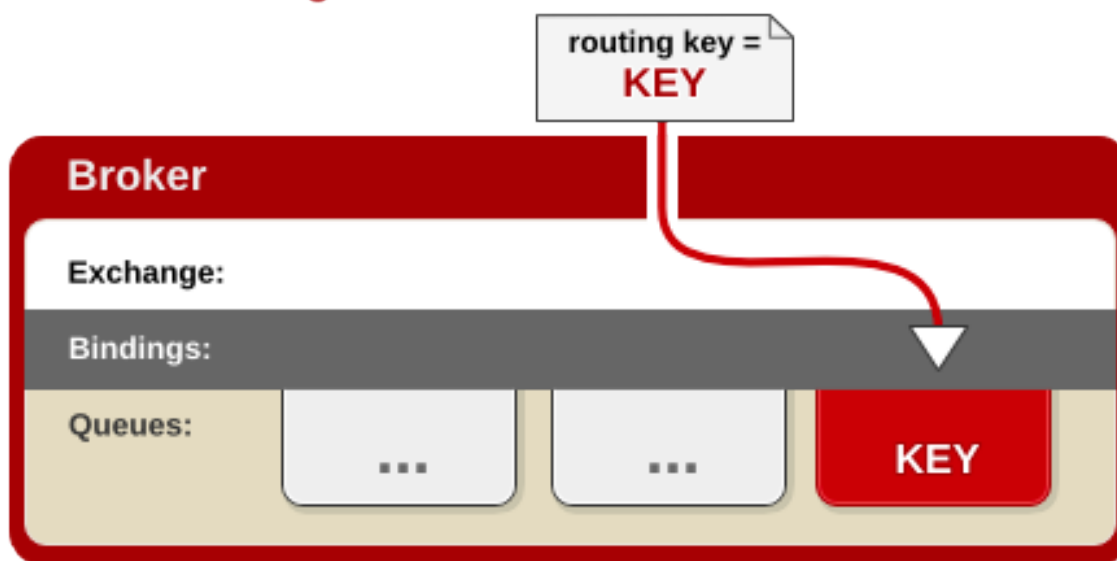


A Fanout exchange sends messages to every queue bound to the exchange.

## 1.2. Direct Exchange

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

## Direct Exchange



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

### 1.3. Topic Exchange

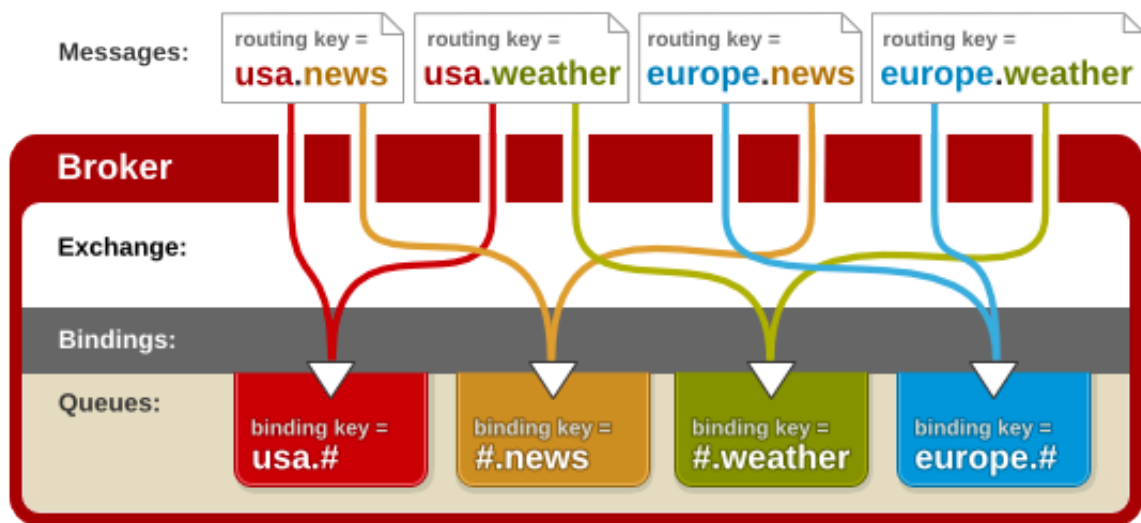
A Topic exchange is similar to a Direct exchange, but uses keys that contain multiple words separated by a "." delimiter. A message producer might create messages with routing keys like **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**.

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

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



## Topic Exchange



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

## 1.4. Custom Exchange Types

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

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



# Examples Overview

This tutorial consists of a series of examples in Python, C++, Java JMS, and C# (for .NET), using the three most commonly used exchange types in MRG Messaging - Direct, Fanout and Topic exchanges. These examples show how to write applications that use the most common messaging paradigms. This chapter contains descriptions of each of the paradigms used throughout the examples. After these examples, there are a few brief sections that show how to enable durable queues and messages, and how to use transactions.

## Overview of the C++, Python, Java JMS, and C# (for .NET) Examples

### Direct

In the direct examples, a message producer writes to the direct exchange, specifying a routing key. A message consumer reads messages from a named queue. A separate configuration program binds the queues, determining which routing keys are associated with each queue. This illustrates clean separation of concerns - message producers need to know only the exchange and the routing key, message consumers need to know only which queue to use on the broker. By changing the bindings in the configuration program, messages can be routed in different ways without affecting message producers or message consumers.

### Fanout

The fanout examples use a fanout exchange and do not use routing keys. Each binding specifies that all messages for a given exchange should be delivered to a given queue.

### Publish/Subscribe

In the publish/subscribe examples, a publisher application writes messages to an exchange, specifying a multi-part key. A subscriber application subscribes to messages that match the relevant parts of these keys, using a private queue for each subscription.

### Request/Response

In the request/response examples, a simple service accepts requests from clients and sends responses back to them. Clients create their own private queues and corresponding routing keys. When a client sends a request to the server, it specifies its own routing key in the **reply-to** field of the request. The server uses the client's **reply-to** field as the routing key for the response.

### Durability

MRG Messaging provides guaranteed delivery unless the server crashes. Durable queues and durable messages are stored using persistent storage on the server, and restored when the broker is restarted, providing guaranteed delivery even if there is a crash. This section shows how to make queues and messages durable.

### Transactions

This section shows how to use local server transactions, which buffer published messages and acknowledgements and process them upon commit, guaranteeing that they will all succeed or fail as a unit.

### XML Exchange

This section shows how to declare an XML Exchange and use it to route XML messages based on their content, using XQuery. Because the XML Exchange can not currently be declared in a pure Java JMS or C# (for .NET) program, this example is shown only for C++ and Python clients. (Java JMS programs can declare an exchange and an XQuery binding using Python or C++, then

use the corresponding exchange and queues using Java JMS. There is also a low level Java API which supports custom exchanges, but this API is not yet documented.)

# Installing MRG Messaging

In order to install MRG Messaging you will need to have registered your system with [Red Hat Network](#)<sup>1</sup>. This table lists the Red Hat Enterprise MRG channels available on Red Hat Network for MRG Messaging.

Channel Name	Operating System	Architecture
Red Hat MRG Messaging	RHEL-4 AS	32bit, 64bit
Red Hat MRG Messaging	RHEL-4 ES	32bit, 64bit
Red Hat MRG Messaging	RHEL-5 Server	32bit, 64bit
Red Hat MRG Messaging	Non-Linux	32bit
Red Hat MRG Messaging Base	RHEL-4 AS	32bit, 64bit
Red Hat MRG Messaging Base	RHEL-4 ES	32bit, 64bit
Red Hat MRG Messaging Base	RHEL-5 Server	32bit, 64bit

Table 3.1. Red Hat Enterprise MRG Channels Available on Red Hat Network



## Important

Before you install Red Hat Enterprise MRG check that your hardware and platform is supported. A complete list is available on the [Red Hat Enterprise MRG Supported Hardware Page](#)<sup>2</sup>.

## 3.1. Installing MRG Messaging on Red Hat Enterprise Linux 5

1. Install the MRG Messaging group using the **yum** command.

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

2. You can check the installation location and that the components have been installed successfully by using the **rpm -ql** command with the name of the package you installed. For example:

```
# rpm -ql rhm
/etc/qpidd.conf
/usr/lib/qpiddlibbdbstore.so.0
...
```



## Note

If you find that yum is not installing all the dependencies you require, make sure that you have registered your system with [Red Hat Network](#)<sup>3</sup>.

<sup>1</sup> <https://rhn.redhat.com/help/about.pxt>

## 3.2. Installing MRG Messaging on Red Hat Enterprise Linux 4

1. Install the MRG Messaging components using the **up2date** command.

```
# up2date python-qpid qpid-java-client qpidc-devel rhm rhm-docs
```

2. You can check the installation location and that the components have been installed successfully by using the **rpm -ql** command with the name of the package you installed. For example:

```
# rpm -ql rhm
/etc/rhmd.conf
/usr/lib/qpid/libbdbstore.so.0
/usr/lib/qpid/libbdbstore.so.0.1.0
...
```



### Note

If you find that **up2date** is not installing all the dependencies you require, make sure that you have registered your system with [Red Hat Network](#)<sup>4</sup>.

## 3.3. Starting the Broker

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

```
# /usr/sbin/qpid -t
[date] [time] info Loaded Module: libbdbstore.so.0
[date] [time] info Locked data directory: /var/lib/qpid
[date] [time] info Management enabled
[date] [time] info Listening on port 5672
```

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

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

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

3. For production use, MRG Messaging is usually run as a service. To start the broker as a service, run the following command as the root user:

```
# service qpid start
Starting qpid daemon:      [ OK ]
```

4. You can check on the status of the service using the **service status** command and stop the broker with **service stop**.

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

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

**Note**

For more detail on downloading, installing and starting the broker, including troubleshooting information, refer to the *MRG Messaging Installation Guide*





# Using MRG Messaging with Python

This section shows how to write direct, fanout, publish/subscribe, request/response, and XML-based routing programs in Python. These concepts are explained in [Chapter 2, Examples Overview](#). It then shows how to use important features like persistence and transactions with MRG Messaging. This chapter does not try to teach the entire MRG Messaging Python API, and it is not encyclopedic in its coverage of AMQP. For more detailed information on the Python API for MRG Messaging, use **pydoc**. For instance, to see all available classes, use the command:

```
$ pydoc qpid
```

To see the methods for the **session** object, use the command:

```
$ pydoc qpid.session
```

For more detailed information on the AMQP model, see the AMQP specification at <http://www.amqp.org>.

The instructions in this section assume you have installed the client libraries and started a broker using the instructions shown in [Chapter 3, Installing MRG Messaging](#).

## 4.1. Creating and Closing Sessions

All of the examples in this section use the same code to initialize the program, create a session, and clean up before exiting. They also use the same imported modules. The following skeleton can be used as the basis to write a wide variety of MRG Messaging applications in Python.

```
import qpid
import sys
import os
from qpid.util import connect
from qpid.connection import Connection
from qpid.datatypes import Message, RangedSet, uuid4
from qpid.queue import Empty

# additional imports for a given example go here

#----- Functions and Classes -----

# Any functions and classes needed for a given example
# go here.

#----- Initialization -----

# Set parameters for login

host="127.0.0.1"
port=5672
user="guest"
password="guest"
```

```
# Create a connection and a session. The constructor for a session
# requires a UUID to uniquely identify the session.

socket = connect(host, port)
connection = Connection (sock=socket, username=user, password=password)
connection.start()
session = connection.session(str(uuid4()))

#----- Main Body of Program -----

#   Main body of each example goes here

#----- Cleanup -----

# Close the session before exiting so there are no open threads.
session.close(timeout=10)
```

### 4.2. Writing Direct Applications in Python.

The following programs work together to implement direct messaging using a Direct exchange:

- **declare\_queues.py** creates a queue on the broker, then exits.
- **direct\_producer.py** publishes messages to the direct exchange.
- **direct\_consumer.py** reads messages from the queue.
- **listener.py** reads messages from the queue using a listener.

#### 4.2.1. Running the Direct Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/python/direct`. To run these programs, do the following:

1. Make sure that a qpidd broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. Declare a message queue and bind it to an exchange by running **declare\_queues.py**, as follows:

```
$ python declare_queues.py
```

This program has no output. After this program has been run, all messages sent to the **amq.direct** exchange using the routing key **routing\_key** are sent to the queue named **message\_queue**.

3. Publish a series of messages to the **amq.direct** exchange by running **direct\_producer.py**, as follows:

```
$ python direct_producer.py
```

This program has no output; the messages are routed to the message queue, as instructed by the binding.

4. Read the messages from the message queue using **direct\_consumer.py** or **listener.py**, as follows:

```
$ python direct_consumer.py
```

or

```
$ python listener.py
```

You should see the following output:

```
message 0
message 1
message 2
message 3
message 4
message 5
message 6
message 7
message 8
message 9
That's all, folks!
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 4.1, “Creating and Closing Sessions”](#).

### 4.2.2. Declaring and Binding a Queue

**declare\_queues.py** creates a queue on the broker. The main body of this program consists of only two lines of code. The first line creates the queue and names it **message\_queue**. The second line determines which messages are routed to the queue, by instructing the broker to route all messages sent to the **amq.direct** exchange with the routing key **routing\_key** to the queue named **message\_queue**.

```
session.queue_declare(queue="message_queue")
session.exchange_bind(exchange="amq.direct", queue="message_queue",
    binding_key="routing_key")
```

### 4.2.3. Publishing Messages to a Direct Exchange

`direct_producer.py` publishes a series of messages to the `amq.direct` exchange. It uses a simple loop to create ten messages, then signals that no more messages are expected by publishing a message with the content "That's all, folks!". The routing key is specified in the delivery properties for the message. Here is the main body for this program.

```
# Create some messages and put them on the broker.
props = session.delivery_properties(routing_key="routing_key")

for i in range(10):
    session.message_transfer(destination="amq.direct",
                             message=Message(props, "message " + str(i)))

session.message_transfer(destination="amq.direct",
                          message=Message(props, "That's all, folks!"))
```

Note that the last message sent is **That's all, folks!**. The consumer looks for this text to determine when all messages have been received.

### 4.2.4. Reading Messages from the Queue

`direct_consumer.py` creates a local queue, subscribes it to the message queue on the server, reads messages, and prints them out. We start by creating a local client queue using `session.incoming()`:

```
local_queue_name = "local_queue"
local_queue = session.incoming(local_queue_name)
```

Next, we subscribe this queue to the server-side queue named `message_queue` and call `start()` to begin message delivery:

```
session.message_subscribe(queue="message_queue",
                           destination=local_queue_name)
local_queue.start()
```

Finally, we read the messages from the local queue, acknowledging each message so it can be removed from the server-side queue:

```
final = "That's all, folks!"    # In a message body, signals the last
                                # message
content = ""                    # Content of the last message read

message = None
while content != final:
    message = local_queue.get(timeout=10)
    content = message.body
    session.message_accept(RangedSet(message.id)) # acknowledge message
    receipt
    print content
```

### 4.2.5. Reading Messages from a Queue using a Listener

**listener.py** receives messages using a message listener. The program provides a method that is called whenever a message is received. Here is the listener class:

```
class Receiver:
    def __init__ (self):
        self.finalReceived = False

    def isFinal (self):
        return self.finalReceived

    def Handler (self, message):
        content = message.body
        session.message_accept(RangedSet(message.id))
        print content
        if content == "That's all, folks!":
            self.finalReceived = True
```

To use this class in our program, we will register the Handler method with a local queue so that it is called whenever a new message is transferred to this queue. First, we must subscribe the local queue to the server-side queue, as we did in the previous section:

```
local_queue_name = "local_queue"
local_queue = session.incoming(local_queue_name)

session.message_subscribe(queue="message_queue",
    destination=local_queue_name)
local_queue.start()
```

Once this is done, we create a receiver and register the Handler method as a message listener for the local queue:

```
receiver = Receiver()
local_queue.listen (receiver.Handler)
```

The Handler method acknowledges each message and prints it out when it is received. It also looks for the final message and signals that it is finished by setting **self.finalReceived** to **true**. Any Python callable that is called with one Message as a parameter may be used as a message listener callback. Now the code that instructs the handler to finish:

```
# Add this to the imports in the skeleton
from time import sleep

# Wait for the receiver to signal that it is done.

while not receiver.isFinal() :
    sleep (1)
```

## 4.3. Writing Fanout Applications in Python

The following programs work together to implement a fanout pattern, where exchanges deliver messages to all queues bound to the exchange.

- `declare_queues.py` creates a queue on the broker, binding it to the fanout exchange.
- `fanout_producer.py` publishes messages to the fanout exchange.
- `fanout_consumer.py` reads messages from the queue.
- `listener.py` reads messages from the queue using a listener.

### 4.3.1. Running the Fanout Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/python/fanout`. To run these programs, do the following:

1. Make sure that a `qpidd` broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the `qpidd` process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In separate windows, start two or more fanout consumers or fanout listeners as follows:

```
$ python fanout_consumer.py
```

or

```
$ python listener.py
```

These programs each create a private queue, bind it to the `amq.fanout` exchange, and wait for messages to arrive on their queue.

3. In a separate window, publish a series of messages to the `amq.fanout` exchange by running `fanout_producer.py`, as follows:

```
$ python fanout_producer.py
```

This program has no output; the messages are routed to the message queue, as instructed by the binding.

4. Go to the windows where you are running consumers or listeners. You should see the following output for each listener or consumer:

```
message 0
message 1
message 2
message 3
message 4
```

```

message 5
message 6
message 7
message 8
message 9
That's all, folks!

```

Now let's take a look at the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 4.1, “Creating and Closing Sessions”](#).

### 4.3.2. Consuming from a Fanout Exchange

**fanout\_consumer.py** creates a private queue, binds it to the fanout exchange, and reads messages delivered to that queue. If multiple instances of **fanout\_consumer.py** are run, each one has its own private queue. Since each session has a unique session name, using the session name as the name of the server-side queue guarantees that it is unique:

```

server_queue_name = session.name
session.queue_declare(queue=server_queue_name)
session.exchange_bind(queue=server_queue_name, exchange="amq.fanout")

```

It then creates a local queue and subscribes it to the server-side queue. Unlike the server-side queue, there is no need to use a globally unique name, since the name of the local queue is meaningful only within the local session. We call the `start()` method to begin delivery to the local queue:

```

local_queue_name = "local_queue"
local_queue = session.incoming(local_queue_name)

session.message_subscribe(queue=server_queue_name,
    destination=local_queue_name)
local_queue.start()

```

Now we read messages from the local queue, finishing when we receive a message that contains the string “That's all, folks!”:

```

# Initialize 'final' and 'content', variables used to identify the last
# message.
final = "That's all, folks!" # In a message body, signals the last
# message
content = "" # Content of the last message read

# Read the messages - acknowledge each one
message = None
while content != final:
    message = local_queue.get(timeout=10)
    content = message.body
    session.message_accept(RangedSet(message.id))
print content

```

### 4.3.3. Publishing Messages to the Fanout Exchange

The message producer publishes its messages to the **amq.fanout** exchange. There is no need for a routing key, but it will still be shown in message logs and on the message received by a client, where it can be useful for identifying the sender for debugging purposes.

```
delivery_properties =
    session.delivery_properties(routing_key="routing_key")

for i in range(10):
    session.message_transfer(destination="amq.fanout",
        message=Message(delivery_properties, "message " + str(i)))

session.message_transfer(destination="amq.fanout",
    message=Message(delivery_properties, "That's all, folks!"))
```

## 4.4. Writing Publish/Subscribe Applications in Python

This section describes two sample programs that implement a Publish/Subscribe application using a topic exchange. Topic exchanges deliver messages based on multi-part routing keys and binding keys that may contain wildcards.

- **topic\_publisher.py** publishes messages to the topic exchange.
- **topic\_subscriber.py** reads messages from the queue.

In this example, the publisher creates messages for topics like news, weather, and sports that happen in regions like Europe, Asia, or the United States. A given consumer may be interested in all weather messages, regardless of region, or it may be interested in news and weather for the United States, but uninterested in items for other regions. In this example, each consumer sets up its own private queues, which receive precisely the messages that particular consumer is interested in.

### 4.4.1. Running the Publish-Subscribe Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/python/pubsub**. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In separate windows, start one or more topic subscribers by running **topic\_subscriber.py**, as follows:

```
$ python topic_subscriber.py
```

You will see output similar to this:

```
Queues created - please start the topic producer
```



```

Subscribing local queue 'local_news' to news-53408183-
fcee-4b92-950b-90abb297e739'
Subscribing local queue 'local_weather' to weather-53408183-
fcee-4b92-950b-90abb297e739'
Subscribing local queue 'local_usa' to usa-53408183-
fcee-4b92-950b-90abb297e739'
Subscribing local queue 'local_europe' to europe-53408183-
fcee-4b92-950b-90abb297e739'
Messages on 'news' queue:

```

Each topic consumer creates a set of private queues, and binds each queue to the **amq.topic** exchange together with a binding that indicates which messages should be routed to the queue.

3. In another window, start the topic publisher, which publishes messages to the **amq.topic** exchange, as follows:

```
$ python topic_publisher.py
```

This program has no output; the messages are routed to the message queues for each `topic_consumer` as specified by the bindings the consumer created.

4. Go back to the window for each topic consumer. You should see output like this:

```

Messages on 'news' queue:
usa.news 0
usa.news 1
usa.news 2
usa.news 3
usa.news 4
europe.news 0
europe.news 1
europe.news 2
europe.news 3
europe.news 4
That's all, folks!
Messages on 'weather' queue:
usa.weather 0
usa.weather 1
usa.weather 2
usa.weather 3
usa.weather 4
europe.weather 0
europe.weather 1
europe.weather 2
europe.weather 3
europe.weather 4
That's all, folks!
Messages on 'usa' queue:
usa.news 0
usa.news 1
usa.news 2

```

```
usa.news 3
usa.news 4
usa.weather 0
usa.weather 1
usa.weather 2
usa.weather 3
usa.weather 4
That's all, folks!
Messages on 'europe' queue:
europe.news 0
europe.news 1
europe.news 2
europe.news 3
europe.news 4
europe.weather 0
europe.weather 1
europe.weather 2
europe.weather 3
europe.weather 4
That's all, folks!
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 4.1, “Creating and Closing Sessions”](#).

### 4.4.2. The Topic Publisher

**topic\_publisher.py** publishes messages to the topic exchange, providing multi-part routing keys like **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**. The publisher has no idea what bindings have been made by subscribers, it simply sends its messages to the topic exchange.

This program defines a function that sends a set of five messages to the topic exchange, using the same routing key for each:

```
def send_msg(routing_key):
    props = session.delivery_properties(routing_key=routing_key)
    for i in range(5):
        session.message_transfer(destination="amq.topic",
                                message=Message(props, routing_key + " " + str(i)))
```

In the main body of the program we use this function to send messages with four different routing keys:

```
# usa.news
send_msg("usa.news")

# usa.weather
send_msg("usa.weather")

# europe.news
send_msg("europe.news")
```

```
# europe.weather
send_msg("europe.weather")
```

When we are finished sending these messages, we send a message using the routing key **control**, indicating that we are done:

```
# Signal termination
props = session.delivery_properties(routing_key="control")
session.message_transfer(destination="amq.topic",
    message=Message(props,"That's all, folks!"))
```

### 4.4.3. The Topic Subscriber

**topic\_subscriber.py** sets up its own queues, binding them to the topic exchange with binding keys that identify interesting messages. It sets up queues for **news**, **weather**, **usa**, and **europe**, then binds these to the topic exchange using binding keys that contain wildcards. For instance, the **news** queue is bound using the binding key **#.news**, and the **usa** queue is bound using the binding key **usa.#**. If a message is published to the **amq.topic** exchange using the routing key **usa.news**, it matches both binding keys, and is delivered to both the **usa** and **news** queues.

Since we will be using four queues, print the contents of a queue in a function so that it can be reused:

```
def dump_queue(queue):
    content = ""                # Content of the last message read
    final = "That's all, folks!" # In a message body, signals the last
    message
    message = 0

    while content != final:
        try:
            message = queue.get(timeout=10)
            content = message.body
            session.message_accept(RangedSet(message.id))
            print content
        except Empty:
            print "No more messages!"
            return
```

You can also write a function to subscribe to a queue:

```
def subscribe_queue(server_queue_name, local_queue_name):

    print "Subscribing local queue '" + local_queue_name + "' to " +
    server_queue_name + "'"

    queue = session.incoming(local_queue_name)

    session.message_subscribe(queue=server_queue_name,
    destination=local_queue_name)
    queue.start()
```

```
return queue
```

Because we are using private server-side queues, we need to use unique names for these queues in the main body of the program. We do this using the session name:

```
# declare queues on the server

news = "news-" + session.name
weather = "weather-" + session.name
usa = "usa-" + session.name
europe = "europe-" + session.name

session.queue_declare(queue=news, exclusive=True)
session.queue_declare(queue=weather, exclusive=True)
session.queue_declare(queue=usa, exclusive=True)
session.queue_declare(queue=europe, exclusive=True)
```

Now the queues can be bound using wildcard matching. The message producer uses routing keys that contain multiple words separated by the “.” delimiter: **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**. Binding keys can include wildcard characters: a “#” matches one or more words, a “\*” matches a single word. In this example we use binding keys like **#.news** (all news items) and **usa.#** (all items in the USA) to match these routing keys:

```
# Routing keys may be "usa.news", "usa.weather", "europe.news", or
"europe.weather".

# The '#' symbol matches one component of a multipart name, e.g. "#.news"
matches
# "europe.news" or "usa.news".

session.exchange_bind(exchange="amq.topic", queue=news,
    binding_key="#.news")
session.exchange_bind(exchange="amq.topic", queue=weather,
    binding_key="#.weather")
session.exchange_bind(exchange="amq.topic", queue=usa, binding_key="usa.#")
session.exchange_bind(exchange="amq.topic", queue=europe,
    binding_key="europe.#")
```

When the topic publisher is finished, it sends a message using the **control** routing key. In the topic subscriber, we need to be able to identify the last message published to each queue, so we route the **control** binding queue to all four queues. AMQP guarantees the order of messages posted to a given queue will be maintained, so we know that when we get the final message, we are finished with the queue. Here is the code in **topic\_subscriber.py** that binds the **control** routing key to each queue:

```
# Bind each queue to the 'control' binding key so we know when to stop

session.exchange_bind(exchange="amq.topic", queue=news,
    binding_key="control")
```

```

session.exchange_bind(exchange="amq.topic", queue=weather,
    binding_key="control")
session.exchange_bind(exchange="amq.topic", queue=usa,
    binding_key="control")
session.exchange_bind(exchange="amq.topic", queue=europe,
    binding_key="control")

```

Finally, the topic subscriber creates local queues, subscribes them to its private queues on the server, and dumps the content of each queue to show what messages have arrived:

```

# Subscribe local queues to server queues

local_news = "local_news"
local_weather = "local_weather"
local_usa = "local_usa"
local_europe = "local_europe"

local_news_queue = subscribe_queue(news, local_news)
local_weather_queue = subscribe_queue(weather, local_weather)
local_usa_queue = subscribe_queue(usa, local_usa)
local_europe_queue = subscribe_queue(europe, local_europe)

# Call dump_queue to print messages from each queue

print "Messages on 'news' queue:"
dump_queue(local_news_queue)

print "Messages on 'weather' queue:"
dump_queue(local_weather_queue)

print "Messages on 'usa' queue:"
dump_queue(local_usa_queue)

print "Messages on 'europe' queue:"
dump_queue(local_europe_queue)

```

## 4.5. Writing Request/Response Applications in Python

This section describes two sample programs that implement a Request/Response application using a Direct exchange.

- **server.py** receives messages, converts them to upper case, and sends them back to the original client.
- **client.py** sends requests to the server as messages, receives responses and prints them to the screen.

### 4.5.1. Running the Request/Response Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/python/request-response`. To run these programs, do the following:

1. Make sure that a `qpidd` broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. Run the server.

```
$ python server.py
```

You should see the following output:

```
Request server running - run your client now.
(Times out after 100 seconds ...)
```

3. In a separate window, start a client:

```
$ python client.py
```

You should see the following output:

```
Request: Twas brillig, and the slithy toves
Request: Did gyre and gimble in the wabe.
Request: All mimsy were the borogroves,
Request: And the mome raths outgrabe.
Messages on queue: reply_to:db0f862e-6b36-4e0f-a4b2-ad049eb435ce
Response: TWAS BRILLIG, AND THE SLITHY TOVES
Response: DID GYRE AND GIMBLE IN THE WABE.
Response: ALL MIMSY WERE THE BOROGROVES,
Response: AND THE MOME RATHS OUTGRABE.
No more messages!
```

Now let's take a look at the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 4.1, "Creating and Closing Sessions"](#).

### 4.5.2. The Server Application

Now let's look at the code for these two applications, which are each based on the skeleton shown at [Section 4.1, "Creating and Closing Sessions"](#). In this application, both programs send and receive messages. The server sets up a queue called **request**, and binds it to the **amq.direct** exchange with the binding key **request**. Clients post all requests to the **amq.direct** exchange using the routing key **request**. Each client creates its own private response queue and a corresponding routing key, which it places in the **reply-to** property of each request it writes to the exchange.

**server.py** creates a request queue, which is used for all clients, reads requests from this queue, and sends responses to the client who made each request. Here is the code that creates the request queue and subscribes to it:

```
session.queue_declare(queue="request", exclusive=True)
```

```

session.exchange_bind(exchange="amq.direct", queue="request",
    binding_key="request")

local_queue_name = "local_queue"

session.message_subscribe(queue="request", destination=local_queue_name)

queue = session.incoming(local_queue_name)
queue.start()

```

The server then creates a local queue, using the destination used in the above subscription, and waits for messages to arrive. If a message arrives, it calls the **respond()** function:

```

queue = session.incoming(local_queue_name)

# If we get a message, send it back to the user (as indicated in the
# ReplyTo property)

while True:
    try:
        request = queue.get(timeout=100)
        respond(session, request)
        session.message_accept(RangedSet(request.id))
    except Empty:
        print "No more messages!"
        break;

```

In the **respond** function, the server takes the body of the message, converts it to upper case, and writes the result to the **amq.direct** exchange using a routing key specified by the client using the message's **reply\_to** property, which contains a **routing\_key** property:

```

def respond(session, request):

    # The routing key for the response is the request's reply-to
    # property. The body for the response is the request's body,
    # converted to upper case.

    message_properties = request.get("message_properties")
    reply_to = message_properties.reply_to
    if reply_to == None:
        raise Exception("This message is missing the 'reply_to' property,
            which is required")

    props =
    session.delivery_properties(routing_key=reply_to["routing_key"])
    session.message_transfer(destination=reply_to["exchange"],
        message=Message(props, request.body.upper()))

```

### 4.5.3. The Client Application

**client.py** creates a private queue for the server's responses and binds to it using a unique routing key. To guarantee uniqueness, it uses the session name for both the name of the queue and the routing key::

```
reply_to = "reply_to:" + session.name
session.queue_declare(queue=reply_to, exclusive=True)
session.exchange_bind(exchange="amq.direct", queue=reply_to,
    binding_key=reply_to)
```

It also creates a local queue, from which it reads the server's responses. It subscribes this queue to its private server-side queue and calls **start()** to start receiving messages:

```
local_queue_name = "local_queue"
queue = session.incoming(local_queue_name)

session.message_subscribe(queue=reply_to, destination=local_queue_name)
queue.start()
```

Next, it sends some lines of poetry to the server, one line at a time, using the routing key for its private queue in the reply-to property:

```
lines = ["Twas brillling, and the slithy toves",
        "Did gyre and gimble in the wabe.",
        "All mimsy were the borogroves,",
        "And the mome raths outgrabe."]

# We will use the same reply_to and routing key
# for each message

message_properties = session.message_properties()
message_properties.reply_to = session.reply_to("amq.direct", reply_to)
delivery_properties = session.delivery_properties(routing_key="request")

for line in lines:
    print "Request: " + line
    session.message_transfer(destination="amq.direct",
        message=Message(message_properties, delivery_properties, line))
```

Finally, we call the **dump\_queue()** function to see the responses we have received from the server:

```
]
dump_queue(reply_to)
```

Here is the definition of the **dump\_queue()** function:

```
def dump_queue(queue_name):
    print "Messages on queue: " + queue_name
```



```

message = 0

while True:
    try:
        message = queue.get(timeout=10)
        content = message.body
        session.message_accept(RangedSet(message.id))
        print "Response: " + content
    except Empty:
        print "No more messages!"
        break
    except:
        print "Unexpected exception!"
        break

```

## 4.6. XML-based Routing in Python

The following programs work together to implement XML-based routing using an XML Exchange:

- **declare\_queues.py** creates a queue on the broker, declares an XML Exchange, subscribes the queue to the XML Exchange using an XQuery in the binding, then exits.
- **xml\_producer.py** publishes messages to the XML Exchange.
- **xml\_consumer.py** reads messages from the queue.
- **listener.py** reads messages from the queue using a listener.

### 4.6.1. Running the XML-based Routing Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/python/xml-exchange`. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. Declare an XML exchange and a message queue, then bind the queue to the exchange by running **declare\_queues.py**, as follows:

```
$ python declare_queues.py
```

This program has no output. After this program has been run, all messages sent to the **xml** exchange using the routing key **weather** are sent to the queue named **message\_queue** if they satisfy the conditions specified in the following XQuery, which is used in the binding:

```

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

```

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

3. Publish a series of messages to the **xml** exchange by running **xml\_producer.py**, as follows:

```
$ python xml_producer.py
```

The messages are routed to the message queue, as prescribed by the binding. Each message represents a weather report, such as this one:

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

4. Read the messages from the message queue using **direct\_consumer.py** or **listener.py**, as follows:

```
$ python xml_consumer.py
```

or

```
$ python listener.py
```

You should see the following output:

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

Now we will look at the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 4.1, "Creating and Closing Sessions"](#).

### 4.6.2. Declaring an XML Exchange, Declaring and Binding a Queue

Now we will declare an XML exchange named **xml**, a queue named **message\_queue**, and a binding that routes messages based on an XQuery.

An XML Exchange differs from a direct exchange in two significant ways. The first is that there is no predeclared XML exchange, because it is not an exchange type defined in the AMQP specification. Therefore, you have to declare an XML exchange, whose type is "xml":

```
session.exchange_declare(exchange="xml", type="xml")
```

The second difference is that an XML Exchange uses an XQuery to determine whether to route the message, based on the XML content of the message or message properties, as shown in the following code:

```
session.queue_declare(queue="message_queue")

binding = {}
binding["xquery"] = """
    let $w := ./weather
    return $w/station = 'Raleigh-Durham International Airport (KRDU)'
       and $w/temperature_f > 50
       and $w/temperature_f - $w/dewpoint > 5
       and $w/wind_speed_mph > 7
       and $w/wind_speed_mph < 20 """

session.exchange_bind(exchange="xml", queue="message_queue",
    binding_key="weather", arguments=binding)
```

Any number of bindings can be created for a given binding key, using a different query for each. This makes it possible for many clients to each subscribe to the weather events that interest them.

### 4.6.3. Publishing to an XML Exchange

Publishing to an XML Exchange is very similar to publishing to a direct exchange — you publish to the exchange using a routing key, which the binding associates with an XQuery:

```
props = session.delivery_properties(routing_key="weather")

for i in range(10):
    print report(i)
    session.message_transfer(destination="xml", message=Message(props,
        report(i)))
```

In the above code, **report(i)** is a function that creates the XML messages used in this program. Each XML message represents a simplified weather report:

```
station = ("Raleigh-Durham International Airport (KRDU)",
    "New Bern, Craven County Regional Airport (KEWN)",
    "Boone, Watauga County Hospital Heliport (KTNB)",
    "Hatteras, Mitchell Field (KHSE)")
wind_speed_mph = ( 0, 2, 5, 10, 16, 22, 28, 35, 42, 51, 61, 70, 80 )
temperature_f = ( 30, 40, 50, 60, 70, 80, 90, 100 )
dewpoint = ( 35, 40, 45, 50 )

def pick_one(list, i):
    return str( list [ i % len(list)] )

def report(i):
    return "<weather>"
        + "<station>" + pick_one(station,i) + "</station>"
```

```
        + "<wind_speed_mph>" + pick_one(wind_speed_mph,i) + "</  
wind_speed_mph>"  
        + "<temperature_f>" + pick_one(temperature_f,i) + "</  
temperature_f>"  
        + "<dewpoint>" + pick_one(dewpoint,i) + "</dewpoint>"  
    + "</weather>")
```

### 4.6.4. Reading from the Message Queue

`xml_consumer.py` and `listener.py` simply read from a message queue and print the messages they receive. This code is identical to the code used to do the same in the direct exchange examples. For instance, here is the body of `xml_consumer.py`:

```
local_queue_name = "local_queue"  
local_queue = session.incoming(local_queue_name)  
  
session.message_subscribe(queue="message_queue",  
    destination=local_queue_name)  
local_queue.start()  
  
message = None  
while True:  
    try:  
        message = local_queue.get(timeout=10)  
        session.message_accept(RangedSet(message.id))  
        content = message.body  
        print content  
    except Empty:  
        print "No more messages!"  
        break
```

## 4.7. Durable Queues and Durable Messages in Python

By default, the message queue will remain active in the broker as long as the broker is running, even though the program that created the queue has terminated. Should the broker crash, however, the queue and any messages would be lost. In order to avoid accidental loss as a result of machine failure, both queues and messages can be made durable.

If a queue is durable, the queue survives a server crash, as well as any durable messages that have been placed on the queue. However, a queue may also be declared *autoDelete*, which means the queue is deleted automatically when the last client unsubscribes to the queue or terminates. If a queue is both durable and *autoDelete*, it is still deleted when the last client unsubscribes or terminates. To make a queue durable, specify `durable="true"` when you declare the queue:

```
session.queue_declare(queue="message_queue", durable="true")
```

To make a message durable, specify `delivery_mode=session.delivery_mode.persistent` in the `message_transfer()` function:

```
session.message_transfer(destination="amq.direct", content=request,
    delivery_mode=session.delivery_mode.persistent)
```

## 4.8. Using Transactions in Python

This section shows how to use local server transactions, which buffer published messages and acknowledgements and process them upon commit, guaranteeing that they will all succeed or fail as a unit. You can easily do this by making the session transactional. Once you do this, all message transfers and acknowledgements are queued until a commit or rollback is done on the session. After a commit or rollback, the session remains transactional, so operations continue to be queued until the next commit or rollback.

To make a session transactional, call `tx_select()`:

```
session.tx_select()
```

To commit all operations pending on a transactional session, call `tx_commit()`:

```
session.tx_commit()
```

To roll back all operations pending on a transactional session, call `tx_rollback()`:

```
session.tx_rollback()
```

Transactions are used primarily to ensure that delivery is kept consistent in a messaging system. For instance, if you want to make sure that messages are properly forwarded, you can make a session transactional, subscribe to one queue, and publish received messages to another queue, acknowledging the initial delivery and doing a commit. If you do this, the publish and consume are atomic, and will both succeed or fail as a unit.

## 4.9. Logging in Python client applications

The MRG Messaging Python client library supports logging using the standard Python logging module. The easiest way to do logging is to use the `basicConfig()`, which reports all warnings and errors:

```
from logging import basicConfig
basicConfig()
```

MRG Messaging also provides a convenience method that makes it easy to specify the level of logging desired. For instance, the following code enables logging at the **DEBUG** level:

```
from qpid.log import enable, DEBUG
enable("qpid.io", DEBUG)
```

For more information on Python logging, see <http://docs.python.org/lib/node425.html>. For more information on MRG Messaging logging, use `$ pydoc qpid.log`.



# Using MRG Messaging with C++

This chapter shows how to write direct, fanout, publish/subscribe, request/response, and XML-based routing programs in C++. These concepts are explained in [Chapter 2, Examples Overview](#). It then shows how to use important features like persistence and transactions with MRG Messaging. This chapter does not try to teach the entire MRG Messaging C++ API, and it is not encyclopedic in its coverage of AMQP. For more detailed information on the C++ API for MRG Messaging, see the Doxygen documentation installed at `/usr/share/doc/qpidc-devel-0.2svn/html/index.html` (where *svn* is a number identifying a release). For more detailed information on the AMQP model, see the AMQP specification at <http://www.amqp.org>.

The instructions in this section assume you have installed the client libraries and started a broker using the instructions shown in [Chapter 3, Installing MRG Messaging](#).

Before running the examples, you need to compile the files using the **make** command.

```
$ make filename
```

The binaries created by the **make** command are standard Linux binaries, and can be run from the command line using `./` before the name of the executable:

```
$ ./filename
```

In order to use **make**, you must have write privileges for the working directory. This generally means that you should copy the `/usr/share/doc/rhm-0.3` directory to a place where you can modify the code and create subdirectories as part of the compilation process.



## Note

For more information on downloading, installing and starting the broker refer to the *MRG Messaging Installation Guide*

## 5.1. Creating and Closing Sessions

All of the examples in this section have been written using the Apache Qpid C++ API, which is the C++ API for MRG Messaging. The examples use the same skeleton code to open a connection, create a session, and clean up before exiting:

```
#include <qpid/client/Connection.h>
#include <qpid/client/Session.h>

#include <unistd.h>
#include <cstdlib>
#include <iostream>

using namespace qpid::client;
using namespace qpid::framing;
```

```
using std::string;

int main(int argc, char** argv) {

    char * host = "127.0.0.1";
    int port = 5672;

    Connection connection;

    try {
        connection.open(host, port);
        Session session = connection.newSession();

        //----- Main body of program
        -----

        //-----

        connection.close();
        return 0;
    } catch(const std::exception& error) {
        std::cout << error.what() << std::endl;
    }
    return 1;
}
```

Use a `ConnectionSettings` object if you need more control over the parameters used in a connection.

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

using namespace qpid::client;

int main(int , char** ) {

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

    Connection connection;
    try {
        connection.open(connectionSettings);
```



```
Session session = connection.newSession();
...
```

## 5.2. Writing Direct Applications in C++

This section describes three programs that implement direct messaging using a Direct exchange:

- **declare\_queues.cpp** binds a queue to the `amq.direct` exchange, so that messages sent to the direct exchange with the routing key value **routing\_key** are delivered to the queue named **message\_queue**.
- **direct\_producer.cpp** publishes messages to the `amq.direct` exchange, using the routing key **routing\_key**.
- **listener.cpp** uses a message listener to receive messages from the queue named **message\_queue**.

### 5.2.1. Running the Direct Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/cpp/direct`. To run these programs, do the following:

1. Make sure that a `qpidd` broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **direct** directory, build the examples using `make`.

```
$ make
```

3. Declare a message queue and bind it to an exchange by running **declare\_queues**, as follows:

```
$ ./declare_queues
```

This program has no output. After this program has been run, all messages sent to the **amq.direct** exchange using the routing key “`routing_key`” are sent to the queue named “`message_queue`”.

4. Publish a series of messages to the **amq.direct** exchange by running **direct\_producer**, as follows:

```
$ ./direct_producer
```

This program has no output; the messages are routed to the message queue, as instructed by the binding.

5. Read the messages from the message queue using **direct\_consumer** or **listener**, as follows:

```
$ ./direct_consumer
```

or

```
$ ./listener
```

You should see the following output:

```
Message: Message 0
Message: Message 1
Message: Message 2
Message: Message 3
Message: Message 4
Message: Message 5
Message: Message 6
Message: Message 7
Message: Message 8
Message: Message 9
Message: That's all, folks!
Shutting down listener for message_queue
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 5.2.2. Declaring and Binding a Queue

The first program in the direct example, **declare\_queues.cpp**, adds only a few lines to the basic skeleton. It creates a queue named **message\_queue**, then binds it to the **amq.direct** exchange using the binding key **routing\_key**.

```
session.queueDeclare(arg::queue="message_queue");
session.exchangeBind(arg::exchange="amq.direct",
    arg::queue="message_queue", arg::bindingKey="routing_key");
```

The queue created by this program continues to exist after the configuration program exits, and any message whose routing key matches the key specified in the binding will be routed to the corresponding queue by the broker.

Note that the configuration program could easily bind other queues using the same routing key, so that multiple queues would receive the same message, or change the queues to which messages with a given routing key are sent. This can be done without affecting producers or consumers.

### 5.2.3. Publishing Messages to a Direct Exchange

The second program in the direct example, **direct\_producer.cpp**, publishes messages to the **amq.direct** exchange using the routing key **routing\_key**. This program uses the **Message** class and **std::stringstream**, so we'll add the following includes to the skeleton:

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

```
#include <sstream>
using std::stringstream;
```

First, create a message and set a routing key. The same routing key will be used for each message we send, so you only need to set this property once.

```
Message message;
message.getDeliveryProperties().setRoutingKey("routing_key");
```

Now send some messages:

```
for (int i=0; i<10; i++) {
    stringstream message_data;
    message_data << "Message " << i;

    message.setData(message_data.str());
    session.messageTransfer(arg::content=message,
        arg::destination="amq.direct");
}
```

Send a final message to indicate termination.

```
message.setData("That's all, folks!");
session.messageTransfer(arg::content=message,
    arg::destination="amq.direct");
```

### 5.2.4. Reading Messages from the Queue

The third program in the direct example, **listener.cpp**, is a message listener that receives messages from a queue. It uses both the **Message**, **MessageListener**, and **SubscriptionManager** classes, so we'll add the following includes to the skeleton:

```
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>
```

To create a message listener, create a class derived from **MessageListener**, and override the **received** method, providing the code that should be executed when a message is received. This listener uses a subscription manager, part of the MRG Messaging library, to subscribe to and receive messages from a queue.

```
class Listener : public MessageListener {
private:
    SubscriptionManager& subscriptions;
public:
    Listener(SubscriptionManager& subscriptions);
    virtual void received(Message& message);
```

```
};
```

Define a constructor that initializes the listener's subscriptions.

```
Listener::Listener(SubscriptionManager& subs) : subscriptions(subs)
{}
```

The main body of the program creates a subscription manager for the session; creates a listener for the subscription; subscribes the subscription manager to a message queue; and runs the subscription manager to receive messages from the queue.

```
SubscriptionManager subscriptions(session);

// Create a listener and subscribe it to the queue named "message_queue"
Listener listener(subscriptions);
subscriptions.subscribe(listener, "message_queue");

// Deliver messages until the subscription is cancelled
// by Listener::received()
subscriptions.run();
```

The MessageListener's `received()` function is called whenever a message is received. In this example the message is printed and tested to see if it is the final message. Once the final message is received, the subscription to the queue is cancelled.

```
void Listener::received(Message& message) {
    std::cout << "Message: " << message.getData() << std::endl;
    if (message.getData() == "That's all, folks!") {
        std::cout << "Shutting down listener for " <<
            message.getDestination()
                << std::endl;
        subscriptions.cancel(message.getDestination());
    }
}
```

## 5.3. Writing Fanout Applications in C++

This section describes two programs that illustrate the use of a Fanout exchange.

- **listener.cpp** makes a unique queue private for each instance of the listener, and binds that queue to the fanout exchange. All messages sent to the fanout exchange are delivered to each listener's queue.
- **fanout\_producer.cpp** publishes messages to the fanout exchange. It does not use a routing key, which is not needed by the fanout exchange.

### 5.3.1. Running the Fanout Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/cpp/fanout`. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **fanout** directory, build the examples using **make**.

```
$ make
```

3. In separate windows, start two or more fanout listeners as follows:

```
$ ./listener
```

The listener creates a private queue, binds it to the **amq.fanout** exchange, and waits for messages to arrive on the queue. When the listener starts, you will see the following message:

```
Listening
```

4. In a separate window, publish a series of messages to the **amq.fanout** exchange by running **fanout\_producer**, as follows:

```
$ ./fanout_producer
```

This program has no output; the messages are routed to the message queue, as prescribed by the binding.

5. Go to the windows where you are running listeners. You should see the following output for each listener:

```
Message: Message 0
Message: Message 1
Message: Message 2
Message: Message 3
Message: Message 4
Message: Message 5
Message: Message 6
Message: Message 7
Message: Message 8
Message: Message 9
Message: That's all, folks!
Shutting down listener for 5ce43b63-83be-4bd3-8545-c16f94d7febf
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, "Creating and Closing Sessions"](#).

### 5.3.2. Consuming from a Fanout Exchange

The first program in the fanout example, **listener.cpp**, creates a private queue, binds it to the **amq.fanout** exchange, and waits for messages to arrive on the queue, printing them out as they arrive. This program uses the **Message** and **SubscriptionManager** classes, so we'll add the following includes to the skeleton:

```
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>
```

This program uses a Listener class that is identical to the one used in the direct example:

```
class Listener : public MessageListener{
private:
    SubscriptionManager& subscriptions;
public:
    Listener(SubscriptionManager& subscriptions);
    virtual void received(Message& message);
};

Listener::Listener(SubscriptionManager& subs) : subscriptions(subs)
{}

void Listener::received(Message& message) {
    std::cout << "Message: " << message.getData() << std::endl;
    if (message.getData() == "That's all, folks!") {
        std::cout << "Shutting down listener for " <<
message.getDestination()
        << std::endl;
        subscriptions.cancel(message.getDestination());
    }
}
```

The listener creates a private queue to receive its messages and binds it to the fanout exchange:

```
    std::string myQueue=session.getId().getName();
    session.queueDeclare(arg::queue=myQueue, arg::exclusive=true,
arg::autoDelete=true);

    session.exchangeBind(arg::exchange="amq.fanout",
arg::queue=myQueue, arg::bindingKey="my-key");
```

In the last line of the code shown above, the binding key is optional, and has no effect on behavior, but it appears in logs and can be helpful for debugging purposes.

Now we create a listener and subscribe it to the queue:

```
SubscriptionManager subscriptions(session);
Listener listener(subscriptions);
```

```
subscriptions.subscribe(listener, myQueue);
```

After we subscribe to the queues, we call **session.sync()** to ensure that all queues and bindings are in place before we run the subscription manager:

```
session.sync();
```

Then we run the subscription manager. Whenever a message is received, the message listener is invoked to print out the message. The call to **subscriptions.run()** terminates when the message listener calls **subscriptions.cancel()** to indicate that it is finished.

```
std::cout << "Listening" << std::endl;
subscriptions.run();
```

### 5.3.3. Publishing Messages to the Fanout Exchange

The second program in this example, **fanout\_producer.cpp**, writes messages to the fanout exchange. It uses the **Message** and **std::stringstream** classes, so we'll add the following includes to the skeleton:

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

```
#include <sstream>
using std::stringstream;
```

Here is the main body of the program:

```
// Send some messages ...
Message message;

for (int i=0; i<10; i++) {
    stringstream message_data;
    message_data << "Message " << i;

    message.setData(message_data.str());
    session.messageTransfer(arg::content=message,
        arg::destination="amq.fanout");
}

// And send a final message to indicate termination.

message.setData("That's all, folks!");
session.messageTransfer(arg::content=message,
    arg::destination="amq.fanout");
```

## 5.4. Writing Publish/Subscribe Applications in C++

This section describes two programs that implement Publish/Subscribe messaging using a topic exchange.

- **topic\_publisher.cpp** sends messages to the **amq.topic** exchange, using the multipart routing keys **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**.
- **topic\_listener.cpp** creates private queues for **news**, **weather**, **usa**, and **europe**, binding them to the **amq.topic** exchange using bindings that match the corresponding parts of the multipart routing keys.



### Note

The **topic\_listener** program reads only messages published after it begins running. Run **topic\_publisher** after starting **topic\_listener** to send messages to the subscriber.

In this example, the publisher creates messages for topics like news, weather, and sports that happen in regions like Europe, Asia, or the United States. A given consumer may be interested in all weather messages, regardless of region, or it may be interested in news and weather for the United States, but uninterested in items for other regions. In this example, each consumer sets up its own private queues, which receive precisely the messages that particular consumer is interested in.

### 5.4.1. Running the Publish-Subscribe Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/cpp/pub-sub**. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **pub-sub** directory, build the examples using **make**.

```
$ make
```

3. In separate windows, start one or more topic subscribers by running **topic\_listener**, as follows:

```
$ ./topic_listener
```

You will see output similar to this:

```
Declaring queue: usa59ef73a3-6aad-4838-8bd7-15f373cd4cba
Subscribing to queue usa59ef73a3-6aad-4838-8bd7-15f373cd4cba
Declaring queue: europe59ef73a3-6aad-4838-8bd7-15f373cd4cba
Subscribing to queue europe59ef73a3-6aad-4838-8bd7-15f373cd4cba
Declaring queue: news59ef73a3-6aad-4838-8bd7-15f373cd4cba
Subscribing to queue news59ef73a3-6aad-4838-8bd7-15f373cd4cba
Declaring queue: weather59ef73a3-6aad-4838-8bd7-15f373cd4cba
Subscribing to queue weather59ef73a3-6aad-4838-8bd7-15f373cd4cba
```



Listening for messages ...

Each topic listener creates a set of private queues, and binds each queue to the **amq.topic** exchange together with a binding that indicates which messages should be routed to the queue.

4. In another window, start the topic publisher, which publishes messages to the **amq.topic** exchange, as follows:

```
$ ./topic_publisher
```

This program has no output; the messages are routed to the message queues for each `topic_consumer` as specified by the bindings the consumer created.

5. Go back to the window for each topic listener. You should see output like this:

```
Message 0 from usa59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 0 from news59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 1 from news59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 1 from usa59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 2 from news59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 2 from usa59ef73a3-6aad-4838-8bd7-15f373cd4cba
Message: Message 3 from news59ef73a3-6aad-4838-8bd7-15f373cd4cba
...
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 5.4.2. Publishing Messages to a Topic Exchange

The first program in the publish/subscribe example, **topic\_publisher.cpp**, defines two new functions: one that publishes messages to the topic exchange, and one that indicates that no more messages are coming. This program uses the **Message** class and **std::stringstream**, so we'll add the following includes to the skeleton:

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

#include <sstream>
using std::stringstream;
```

The **publish\_messages()** function publishes a series of five messages using the specified routing key.

```
void publish_messages(Session& session, string routing_key)
{
    Message message;

    // Set the routing key once, we'll use the same routing key for all
    // messages.
```

```
message.getDeliveryProperties().setRoutingKey(routing_key);
for (int i=0; i<5; i++) {
    stringstream message_data;
    message_data << "Message " << i;

    message.setData(message_data.str());
    session.messageTransfer(arg::content=message,
arg::destination="amq.topic");
}
}
```

The `no_more_messages()` function signals the end of messages using the control routing key, which is reserved for control messages.

```
void no_more_messages(Session& session)
{
    Message message;

    message.getDeliveryProperties().setRoutingKey("control");
    message.setData("That's all, folks!");
    session.messageTransfer(arg::content=message,
arg::destination="amq.topic");
}
```

In the main body of the program, messages are published using four different routing keys, and then the end of messages is indicated by a message sent to a separate routing key.

```
publish_messages(session, "usa.news");
publish_messages(session, "usa.weather");
publish_messages(session, "europe.news");
publish_messages(session, "europe.weather");

no_more_messages(session);
```

### 5.4.3. Reading Messages from the Queue

The second program in the publish/subscribe example, `topic_listener.cpp`, creates a local private queue, with a unique name, for each of the four binding keys it specifies: `usa.#`, `europe.#`, `#.news`, and `#.weather`, and creates a listener. This program uses the `Message`, `MessageListener`, and `SubscriptionManager` classes, so we'll add the following includes to the skeleton:

```
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>
```

To make it easier to manage our queue subscriptions, we can write a listener that can listen to multiple queues. Here is the class declaration for the listener:

```

class Listener : public MessageListener {
private:
    Session& session;
    SubscriptionManager subscriptions;
public:
    Listener(Session& session);
    virtual void prepareQueue(std::string queue, std::string routing_key);
    virtual void received(Message& message);
    virtual void listen();
    ~Listener() { };
};

```

This listener is the heart of the program, and makes it possible for the rest of the program to be extremely simple. It uses a subscription manager to keep track of the queue subscriptions. The subscription manager constructor takes the session as an argument. It is initialized in the listener's constructor:

```

Listener::Listener(Session& session) :
    session(session),
    subscriptions(session)
{
}

```

In this example, we use the subscription manager to subscribe to multiple queues, then run the subscription manager to receive messages from all subscribed queues. The `prepareQueue()` method creates a queue using a queue name and a routing key supplied as arguments:

```

listener.prepareQueue("usa", "usa.#");

```

The `prepareQueue()` method combines the requested queue name with the the session ID to guarantee that the queue name is unique and will not clash with queues used by other clients that subscribe to a given topic. It then subscribes to the queue in the subscription manager. The code for `prepareQueue()` is:

```

void Listener::prepareQueue(std::string queue, std::string routing_key) {

    /* Create a unique queue name for this consumer by concatenating
     * the queue name parameter with the Session ID.
     */

    queue += session.getId().str();
    std::cout << "Declaring queue: " << queue << std::endl;

    /* Declare an exclusive queue on the broker
     */

    session.queueDeclare(arg::queue=queue, arg::exclusive=true,
arg::autoDelete=true);

    /* Route messages to the new queue if they match the routing key.

```

```

*
* Also route any messages to with the "control" routing key to
* this queue so we know when it's time to stop. A publisher sends
* a message with the content "That's all, Folks!", using the
* "control" routing key, when it is finished.
*/

session.exchangeBind(arg::exchange="amq.topic", arg::queue=queue,
arg::bindingKey=routing_key);
session.exchangeBind(arg::exchange="amq.topic", arg::queue=queue,
arg::bindingKey="control");

/*
* subscribe to the queue using the subscription manager.
*/

std::cout << "Subscribing to queue " << queue << std::endl;
subscriptions.subscribe(*this, queue);
}

```

The listener's **listen()** method listens to all subscribed queues by running the subscription manager:

```

void Listener::listen() {
    subscriptions.run();
}

```

When a message is received, it will be printed. If the message signals termination, we cancel the subscription to the queue the message came from. If no more subscriptions exist in the subscription manager, we also stop the subscription manager to return control to the application:

```

void Listener::received(Message& message) {
    std::cout << "Message: " << message.getData() << " from " <<
message.getDestination() << std::endl;

    if (message.getData() == "That's all, folks!") {
        std::cout << "Shutting down listener for " <<
message.getDestination() << std::endl;
        subscriptions.cancel(message.getDestination());
    }
}

```



### Note

In the Java and Python bindings, messages need to be explicitly acknowledged. In C++, by default this is not needed, although the acknowledgement policy can be set to require explicit message acknowledgement.

Now that we have the code for the listener, the code for the main program is simplified:

```
// Create a listener for the session

Listener listener(session);

// Subscribe to messages on the queues we are interested in

listener.prepareQueue("usa", "usa.#");
listener.prepareQueue("europe", "europe.#");
listener.prepareQueue("news", "#.news");
listener.prepareQueue("weather", "#.weather");

// Wait for the broker to indicate that our queues have been created.
session.sync();

std::cout << "Listening for messages ..." << std::endl;

// Give up control and receive messages
listener.listen();
```

## 5.5. Writing Request/Response Applications in C++

In the request/response examples, we write a server that accepts strings from clients and converts them to upper case, sending the result back to the requesting client. This example consists of two programs.

- **client.cpp** is a client application that sends messages to the server.
- **server.cpp** is a service that accepts messages, converts their content to upper case, and sends the result to the **amq.direct** exchange, using the request's **reply-to** property as the routing key for the response.



### Note

Start **server** before you start **client**, since the client only works if there is a running server.

Insert the code from each example into the skeleton at [Section 5.1, “Creating and Closing Sessions”](#).

### 5.5.1. Running the Request/Response Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/cpp/request-response**. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **request-response** directory, build the examples using make.

```
$ make
```

3. Run the server.

```
$ ./server
```

You should see the following output:

```
$ ./server
Activating request queue listener for: request
Waiting for requests
```

4. In a separate window, start a client:

```
$ ./client
```

You should see the following output:

```
Activating response queue listener for: clientac27e517-1788-4d87-9a74-
da5cbc34ca51
Request: Twas brillig, and the slithy toves
Request: Did gire and gybble in the wabe.
Request: All mimsy were the borogroves,
Request: And the mome raths outgrabe.
Waiting for all responses to arrive ...
Response: TWAS BRILLIG, AND THE SLITHY TOVES
Response: DID GIRE AND GYMBLE IN THE WABE.
Response: ALL MIMSY WERE THE BOROGROVES,
Response: AND THE MOME RATHS OUTGRABE.
Shutting down listener for clientac27e517-1788-4d87-9a74-da5cbc34ca51
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 5.5.2. The Client Application

The first program in the request-response example, **client.cpp**, sets up a private response queue to receive responses from the server, then sends messages the server, listening to the response queue for the server's responses. This program uses the **Message**, **MessageListener**, and **MessageListener** classes and **std::stringstream**, so we'll add the following includes to the skeleton:

```
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>
```

```
#include <sstream>
using std::stringstream;
```

First, set up the private response queue and subscribe to it:

```
stringstream response_queue;
response_queue << "client" << session.getId().getName();

// Use the name of the response queue as the routing key

session.queueDeclare(arg::queue=response_queue.str());
session.exchangeBind(arg::exchange="amq.direct",
    arg::queue=response_queue.str(), arg::bindingKey=response_queue.str());

// Create a listener for the response queue and listen for response
messages.
std::cout << "Activating response queue listener for: " <<
    response_queue.str() << std::endl;
SubscriptionManager subscriptions(session);
Listener listener(subscriptions);
subscriptions.subscribe(listener, response_queue.str());
```

Set some properties that will be used for all requests. The routing key for a request is **request**. The reply-to property is set to the routing key for the client's private queue.

```
Message request;

request.getDeliveryProperties().setRoutingKey("request");
request.getMessageProperties().setReplyTo(ReplyTo("amq.direct",
    response_queue.str()));
```

Now send some requests...

```
string s[] = {
    "Twas brillig, and the slithy toves",
    "Did gire and gymble in the wabe.",
    "All mimsy were the borogroves,",
    "And the mome raths outgrabe."
};

for (int i=0; i<4; i++) {
    request.setData(s[i]);
    session.messageTransfer(arg::content=request,
        arg::destination="amq.direct");
    std::cout << "Request: " << s[i] << std::endl;
}
```

And wait for responses to arrive:

```
std::cout << "Waiting for all responses to arrive ..." << std::endl;
subscriptions.run();
```

### 5.5.3. The Server Application

The second program in the request-response example, **server.cpp**, uses the **reply-to** property as the routing key for responses. This program uses the **AsyncSession**, **Message**, **MessageListener**, and **SubscriptionManager** classes and **std::stringstream**, so we'll add the following includes to the skeleton:

```
#include <qpid/client/AsyncSession.h>
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>

#include <sstream>
using std::stringstream;
```

The main body of **server.cpp** creates an exclusive queue for requests, then waits for messages to arrive.

```
string request_queue = "request";

session.queueDeclare(arg::queue=request_queue);
session.exchangeBind(arg::exchange="amq.direct", arg::queue=request_queue,
    arg::bindingKey=request_queue);

SubscriptionManager subscriptions(session);
Listener listener(subscriptions, session);
subscriptions.subscribe(listener, request_queue);

subscriptions.run();
```

This program uses a listener similar to those in other programs we have discussed, but with one optimization: it uses an asynchronous session to send messages. An Asynchronous session does not wait for acknowledgements when sending messages, which means it can be much faster. However, using asynchronous sessions introduces some programming issues you should be aware of; these are discussed in [Section 5.9, “Optimizing message transfer with asynchronous sessions in C++”](#). Here is the class declaration for the listener:

```
class Listener : public MessageListener{
private:
    SubscriptionManager& subscriptions;
    AsyncSession asyncSession;
public:
    Listener(SubscriptionManager& subscriptions, Session& session);
    virtual void received(Message& message);
```



```
};
```

And here is the constructor:

```
Listener::Listener(SubscriptionManager& subs, Session& session)
    : subscriptions(subs), asyncSession(session)
{}

```

The listener's **received()** method converts the request's content to upper case, then sends a response to the broker, using the request's **reply-to** property as the routing key for the response.

```
void Listener::received(Message& request) {

    Message response;
    string routingKey;

    if (request.getMessageProperties().hasReplyTo()) {
        routingKey =
request.getMessageProperties().getReplyTo().getRoutingKey();
    } else {
        std::cout << "Error: " << "No routing key for request (" <<
request.getData() << ")" << std::endl;
        return;
    }

    std::cout << "Request:: " << request.getData() << " (" << routingKey <<
")" << std::endl;

    // Transform message content to upper case
    std::string s = request.getData();
    std::transform (s.begin(), s.end(), s.begin(), toupper);
    response.setData(s);

    // Send it back to the user
    response.getDeliveryProperties().setRoutingKey(routingKey);
    asyncSession.messageTransfer(arg::content=response,
arg::destination="amq.direct");

    if ( request.getData() == "That's all, folks!" )
        dispatcher.stop();
}

```

## 5.6. XML-based Routing in C++

The following programs work together to implement XML-based routing using an XML Exchange:

- **declare\_queues.cpp** creates a queue on the broker, declares an XML Exchange, subscribes the queue to the XML Exchange using an XQuery in the binding, then exits.
- **xml\_producer.cpp** publishes messages to the XML Exchange.

- `xml_consumer.cpp` reads messages from the queue.
- `listener.cpp` reads messages from the queue using a listener.

### 5.6.1. Running the XML-based Routing Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/python/xml-exchange`. To run these programs, do the following:

1. Make sure that a `qpidd` broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. Declare an XML exchange and a message queue, then bind the queue to the exchange by running **declare\_queues**, as follows:

```
$ ./declare_queues
```

This program has no output. After this program has been run, all messages sent to the **xml** exchange using the routing key **content\_feed** are sent to the queue named **message\_queue** if they satisfy the conditions specified in the following XQuery, which is used in the binding:

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

This query is true if the message ID is an odd number or if the message has an application header of **control='end'**.

3. Publish a series of messages to the **xml** exchange by running **xml\_producer.py**, as follows:

```
$ ./xml_producer
```

You will see the following output, which shows the messages that are produced:

```
Message data: <message><id>0</id></message>  
Message data: <message><id>1</id></message>  
Message data: <message><id>2</id></message>  
Message data: <message><id>3</id></message>  
Message data: <message><id>4</id></message>  
Message data: <message><id>5</id></message>  
Message data: <message><id>6</id></message>  
Message data: <message><id>7</id></message>  
Message data: <message><id>8</id></message>  
Message data: <message><id>9</id></message>  
>
```

4. Read the messages from the message queue using **listener**, as follows:

```
$ ./listener
```

You should see the following output:

```
Message: <message><id>1</id></message>
Message: <message><id>3</id></message>
Message: <message><id>5</id></message>
Message: <message><id>7</id></message>
Message: <message><id>9</id></message>
Message: <end>That's all, folks!</end>
Shutting down listener for message_queue
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 5.6.2. Declaring an XML Exchange, Declaring and Binding a Queue

Now we will declare an XML exchange named **xml**, a queue named **message\_queue**, and a binding that routes messages based on an XQuery.

An XML Exchange differs from a direct exchange in two significant ways. The first is that there is no predeclared XML exchange, because it is not an exchange type defined in the AMQP specification. Therefore, you have to declare an XML exchange, whose type is **xml**:

```
session.exchangeDeclare(arg::exchange="xml", arg::type="xml");
```

The second difference is that an XML Exchange uses an XQuery to determine whether to route the message, based on the XML content of the message or message properties, as shown in the following code:

```
FieldTable binding;
binding.setString("xquery", "declare variable $control external;"
                          " ./message/id mod 2 = 1 or $control = 'end'");

session.exchangeBind(arg::exchange="xml", arg::queue="message_queue",
                    arg::bindingKey="content_feed", arg::arguments=binding);
```

In this particular example, only one binding uses this binding key, but you can create any number of bindings for a given binding key with the XML Exchange. For instance, a second binding might route even numbered messages to a different queue.

Note that this query has an external variable. Each application message header is bound to an external variable with the same name and value as the header before the routing query is invoked.

### 5.6.3. Publishing to an XML Exchange

The next program, **xml\_producer.cpp**, publishes XML messages to the XML exchange. This program uses the **AsyncSession** and **Message** classes and **std::stringstream**, so we'll add the following includes to the skeleton:

```
#include <qpid/client/AsyncSession.h>
#include <qpid/client/Message.h>

#include <sstream>
using std::stringstream;
```

Publishing to an XML Exchange is very similar to publishing to a direct exchange — you publish to the exchange using a routing key, and any existing bindings for that routing key are used to route the message. When publishing messages in a loop, we cast to an asynchronous session, which does not wait for acknowledgements and is more efficient. This optimization is discussed in [Section 5.9, “Optimizing message transfer with asynchronous sessions in C++”](#).

```
message.getDeliveryProperties().setRoutingKey("content_feed");
message.getHeaders().setString("control", "continue");

// Now send some messages ...

for (int i=0; i<10; i++) {
    stringstream message_data;
    message_data << "<message><id>" << i << "</id></message>";

    std::cout << "Message data: " << message_data.str() << std::endl;

    message.setData(message_data.str());
    async(session).messageTransfer(arg::content=message,
arg::destination="xml");
}

// And send a final message to indicate termination.

message.getHeaders().setString("control", "end");
message.setData("<end>That's all, folks!</end>");
    session.messageTransfer(arg::content=message,
arg::destination="xml");
```

### 5.6.4. Reading from the Message Queue

**listener.cpp** is identical to the listener used for the direct exchange example—it simply reads messages from the message queue. It uses the **Message** and **SubscriptionManager** classes, so we will add these includes to the skeleton:

```
#include <qpid/client/Message.h>
#include <qpid/client/SubscriptionManager.h>
```

Here is the listener class for this program:

```

class Listener : public MessageListener{
private:
    SubscriptionManager& subscriptions;
public:
    Listener(SubscriptionManager& subscriptions);
    virtual void received(Message& message);
};

Listener::Listener(SubscriptionManager& subs) : subscriptions(subs)
{}

void Listener::received(Message& message) {
    std::cout << "Message: " << message.getData() << std::endl;
    if (message.getHeaders().getString("control") == "end") {
        std::cout << "Shutting down listener for " <<
            message.getDestination()
                << std::endl;
        subscriptions.cancel(message.getDestination());
    }
}

```

And here is the main body of the program:

```

SubscriptionManager subscriptions(session);

// Create a listener and subscribe it to the queue named "message_queue"
Listener listener(subscriptions);
subscriptions.subscribe(listener, "message_queue");

// Deliver messages until the subscription is cancelled
// by Listener::received()
subscriptions.run();

```

## 5.7. Durable Queues and Durable Messages in C++

By default, the message queue will remain active in the broker as long as the broker is running, even though the program that created the queue has terminated. Should the broker crash, however, the queue and any messages would be lost. In order to avoid accidental loss as a result of machine failure, both queues and messages can be made durable.

If a queue is durable, the queue survives a server crash, as well as any durable messages that have been placed on the queue (non-durable messages on a durable queue may be lost if the server crashes). However, a queue may also be declared **autoDelete**, which means the queue is deleted automatically when the last client unsubscribes to the queue or terminates. If a queue is both durable and **autoDelete**, it is still deleted when the last client unsubscribes or terminates. To make a queue durable, specify `durable=true` when you declare the queue:

```

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

```

To make a message durable, set the delivery mode for the message to **PERSISTENT**:

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

### 5.8. Using Transactions in C++

This section shows how to use server local transactions, which buffer published messages and acknowledgements and process them upon commit, guaranteeing that they will all succeed or fail as a unit. In AMQP you can do this by making the session transactional. Once you do this, all message transfers and acknowledgements are queued up until a commit or rollback is done on the session. After a commit or rollback, the session remains transactional, so operations continue to be queued up until the next commit or rollback.

To make a session transactional, call `tx_select()`:

```
session.txSelect()
```

To commit all operations pending on a transactional session, call `tx_commit()`:

```
session.txCommit()
```

To roll back all operations pending on a transactional session, call `tx_rollback()`:

```
session.txRollback()
```

Transactions are used primarily to ensure that delivery is kept consistent in a messaging system. For instance, if you want to make sure that messages are properly forwarded, you can make a session transactional, subscribe to one queue, and publish received messages to another queue, acknowledging the initial delivery and doing a commit. If you do this, the publish and consume are atomic, and will both succeed or fail as a unit.

### 5.9. Optimizing message transfer with asynchronous sessions in C++

When doing many message transfers, we can significantly improve our speed by using an asynchronous session. With a synchronous session, every call to `messageTransfer()` waits for confirmation before returning. An asynchronous session assumes that transfers succeed, and returns without waiting for confirmation. In general, applications should use a synchronous session for everything except for message transfer, because asynchronous sessions can cause a variety of rather confusing programming errors, and there is little performance improvement except when doing message transfer. For message transfer, cast the synchronous session to an asynchronous session to improve speed, and use `Session.sync()` to wait for all pending operations to complete and detect any errors before going on to do other operations:

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

for (int i=0; i<10; i++) {
    stringstream message_data;
    message_data << "Message " << i;

    message.setData(message_data.str());
```

```

    async(session).messageTransfer(arg::content=message,
    arg::destination="amq.direct");
}

session.sync();

```

When using asynchronous sessions, many assumptions programmers generally make do not hold. For instance:

- You cannot assume that a command is complete just because the function returns. Issue a `sync()` on the session to be sure all commands are complete.
- You cannot assume the command will succeed just because the function returns without an exception. The exception may arrive later.
- An exception from an `async` command may be thrown by a *later* function call on the session. The exception response from the broker is processed in the background and puts the session in exception mode; all subsequent calls on the session will fail by throwing the exception.

`sync()` allows you to wait for all commands issued so far to complete. You can also test the outcome of individual commands by using the objects they return, see the Doxygen documentation for details. Asynchronous sessions should be used for commands that are performance-critical, and these sections should be thoroughly tested. Synchronous sessions obey the rules most programmers are used to: they return only when the command is complete and throw an exception immediately if the command failed.

## 5.10. Handling Failover in C++ Connections

The MRG Messaging broker can be run in clustering mode, which provides high reliability at-least-once messaging. If one broker in a cluster fails, clients can choose another broker in the cluster and continue their work.

A client designed to work with a cluster needs to respond appropriately when when a connection fails in order to reconnect to another broker in the cluster. MRG Messaging provides several classes to make this easier.

Instead of using a `Connection` object directly, you can use a **FailoverManager** to manage a connection. A `FailoverManager` creates its own connection and sessions, and if a connection fails, it reconnects to another broker in the same cluster.

The `FailoverManager` uses the Command design pattern. In your program, each command is represented by a class derived from **Failover::Command**, and it implements an `execute()` method that can contain any number of messaging operations. Your program calls **FailoverManager::execute()**, passing a `Command` object as a parameter. For instance, if **sender** is an object derived from `Failover::Command`, the following code executes the command:

```

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

ConnectionSettings settings;
FailoverManager connection(settings);
connection.execute(sender);

```

To execute the command, the FailoverManager creates a connection, creates a session for the connection, and calls the sender's **execute()** method; if the connection is failed over to another broker on the cluster, the FailoverManager creates a new connection and session, then calls the sender's **execute()** method again, using a flag to indicate that the method is invoked because of a failover.

Now we will show how to implement a **FailoverManager::Command** for sending or receiving messages.

### 5.10.1. Sending Messages in a FailoverManager::Command

We just showed an example that uses a **sender** object to send messages using a FailoverManager. Here is the class declaration for the sender.

```
#include <qpid/client/FailoverManager.h>
#include <qpid/client/AsyncSession.h>
#include <qpid/client/Message.h>
#include <qpid/client/MessageReplayTracker.h>
#include <qpid/Exception.h>

class Sender : public FailoverManager::Command
{
public:
    Sender(const std::string& queue, uint count);
    void execute(AsyncSession& session, bool isRetry);
    uint getSent();
private:
    MessageReplayTracker sender;
    const uint count;
    uint sent;
    Message message;
};
```

The **execute()** method takes two parameters. The first parameter is the session object created by the FailoverManager. The second parameter is a flag called **isRetry**, which is set to true if the function is being called as a result of a broker failure. Here is the code for the **execute()** function in the **Sender** class.

```
void Sender::execute(AsyncSession& session, bool isRetry)
{
    if (isRetry) sender.replay(session);
    else sender.init(session);
    while (sent < count) {
        stringstream message_data;
        message_data << ++sent;
        message.setData(message_data.str());
        message.getHeaders().setInt("sn", sent);
    }
```



```

        sender.send(message);
        if (count > 1000 && !(sent % 1000)) {
            std::cout << "sent " << sent << " of " << count << std::endl;
        }
    }
    message.setData("That's all, folks!");
    sender.send(message);
}

```

If the **execute()** function is called with **isRetry=false**, this method sends a set of messages. If it is called with **isRetry=true**, it resends any messages that have not been acknowledged. To do this, it uses a **MessageReplayTracker** to send messages. This class tracks message acknowledgements, and can resend all messages that have not been acknowledged using **MessageReplayTracker::replay()**.

### 5.10.2. Receiving Messages with a **FailoverManager::Command**

A **FailoverManager::Command** can also set up a subscription and register a listener for incoming messages.

```
FailoverManager connection(settings, &listener);
```

Here is an example of a listener class designed to be used in failover applications.

```

class Listener : public MessageListener,
                 public FailoverManager::Command,
                 public FailoverManager::ReconnectionStrategy
{
public:
    Listener();
    void received(Message& message);
    void execute(AsyncSession& session, bool isRetry);
    void check();
    void editUrlList(std::vector<Url>& urls);
private:
    Subscription subscription;
    uint count;
    uint skipped;
    uint lastSn;
    bool gaps;
};

```

This message listener is derived from **FailoverManager::Command**, so it is also a command. When invoked as a command, it subscribes to the message queue and waits for messages. If a broker failure has occurred, it prints a message to the screen before resubscribing.

```
void Listener::execute(AsyncSession& session, bool isRetry)
```

```
{
    if (isRetry) {
        std::cout << "Resuming from " << count << std::endl;
    }
    SubscriptionManager subs(session);
    subscription = subs.subscribe(*this, "message_queue");
    subs.run();
}
```

### 5.10.3. Choosing Brokers for Reconnect

In some environments, certain brokers may be preferable. For instance, a broker may be preferred because it is on a local server or available via a faster network.

The **MessageListener** class we saw in the previous section is derived from **FailoverManager::ReconnectionStrategy**, which allows applications to choose which brokers should be tried first when reconnecting after a failure by implementing an **editUrlList()** method. Here is an **editUrlList()** method for our **Listener** class.

```
void Listener::editUrlList(std::vector<Url>& urls)
{
    /**
     * A more realistic algorithm would be to search through the list
     * for preferred hosts and ensure they come first in the list.
     */
    if (urls.size() > 1) std::rotate(urls.begin(), urls.begin() + 1,
    urls.end());
}
```

The **urls** parameter is the set of urls available for reconnection in the cluster. The method can edit this list to specify which brokers should be tried, and in what order.

## 5.11. Using logging in C++

Logging can give you a very detailed look at the interaction between an application and the broker, which is extremely useful for debugging.

The Qpid broker and C++ clients can both use environment variables to enable logging. The man page for qpid lists all available logging options, and shows the corresponding environment variables. (The man page also shows how command line options and configuration files can be used to set logging and other options for the broker.)

In general, use **QPID\_LOG\_ENABLE** to set the level of logging you are interested in (trace, debug, info, notice, warning, error, or critical):

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

Use **QPID\_LOG\_OUTPUT** to determine where logging output should be sent. This is either a file name or the special values **stderr**, **stdout**, or **syslog**:

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



# Using MRG Messaging with Java JMS

This chapter shows how to write direct, publish/subscribe, and request/response programs in Java JMS. These paradigms are explained in [Chapter 2, Examples Overview](#). It then shows how to use important features like persistence and transactions with MRG Messaging. Unlike Python and C++, Java has an established API standard for messaging called JMS (Java Message Service). This chapter does not attempt to teach JMS programming. We suggest that you look at the [Java JMS Tutorial](#)<sup>1</sup> for an extensive tutorial on JMS.

This chapter focuses on using Java JMS that can work together with AMQP clients written in other languages, even if these other clients use features not directly supported in Java JMS. In general, we accomplish this by using JNDI (Java Naming and Directory Interface) to specify properties for configuring queues and bindings.

The instructions in this section assume you have installed the client libraries and started a broker using the instructions shown in [Chapter 3, Installing MRG Messaging](#).

## Running the Examples

The example code shown in this chapter is shipped as part of the **rh-m-docs** package. Use **rpm** to see where this code is installed on your system:

```
$ rpm -qal rh-m-docs
```

By default, these examples are placed in **/usr/share/doc/rh-m-0.3/java**. There is a script in that directory that compiles and runs sample programs; it is called **runSample.sh**. This script takes the name of the class to be run as an argument, for example:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.direct.Producer
```

Run this script in a directory where you have write permission. You might need to copy the entire **/usr/share/doc/rh-m-0.3/java** directory to a place where you can modify the code and create subdirectories as part of the compilation process. If you do not intend to modify the code, you can simply copy the script and run it from a directory where you have write privileges.

## 6.1. Java JMS Client Compatibility and Interoperability

The MRG Messaging Java JMS client library is compatible with both Java JMS and AMQP. However, there are two limitations you will need to keep in mind:

1. Java JMS compatibility is defined only at the source code level, and
2. You will need to follow certain guidelines if you want your programs to work correctly with AMQP clients written in other languages.

Java JMS is an Application Programming Interface (API), not a wire-level messaging standard. Programs written using our Java JMS client library will interoperate with each other and programs written with other AMQP-based Java JMS clients. If you program using only the Java JMS API, your code can also be run on other Java JMS systems if you configure the environment properly using JNDI.

---

<sup>1</sup> [java.sun.com/products/jms/tutorial/](http://java.sun.com/products/jms/tutorial/)

Interoperability with AMQP clients written in other languages is straightforward for simple applications, but there are some issues that you should keep in mind, especially if these clients use features not found in Java JMS:

- To declare AMQP queues and bindings for your Java JMS program, it is generally best to use JNDI. This allows you to do many things not directly supported by the Java JMS API, including specifying routing keys whose name differs from the queue name, binding to a given queue with multiple routing keys, declaring properties for the queue, etc. This is discussed in detail in [Section 6.2, “Creating and Closing Connections and Sessions with JNDI”](#)
- Declare your queues before using them. In Java JMS, a queue is created implicitly if you attempt to read from it, but not if you attempt to publish to it. In AMQP, queues must be explicitly declared.
- Make sure the content of your message will make sense to clients written in other languages. Serialized Java objects or maps are not easily processed in other languages.
- Make sure that the *message-type* of your message is correctly declared. By default, the Java JMS client uses byte messages. Other message types are discussed in the [Java JMS specification](#)<sup>2</sup>.

## 6.2. Creating and Closing Connections and Sessions with JNDI

The examples described in this chapter create and close connections and sessions using a connection factory that is parameterized using the Java Naming and Directory Interface (JNDI). This section shows the code needed to work with JNDI in MRG Messaging programs, then shows the JNDI properties used in MRG Messaging and the format of a Connection URL and a Binding URL.

### 6.2.1. Basic JNDI Programming for MRG Messaging

The following code establishes the initial JNDI context, which represents the JNDI configuration:

```
Hashtable<String, String> jndiEnvironment = new Hashtable<String,
String>();

// set the factory class
jndiEnvironment.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.qpid.jndi.PropertiesFileInitialContextFactory");

// set the connection factory name
jndiEnvironment.put("connectionfactory.ConnectionFactory",
"amqp://username:password@clientid/test?brokerlist='tcp://
localhost:5672'");

// Set the provider URL that points to a property file
jndiEnvironment.put(Context.PROVIDER_URL, "myPROviderURLPath");

// create the initial context
InitialContext initialContext = new InitialContext(jndiEnvironment);
```

Once the JNDI configuration has been created, we can get the connection factory as follows:

```
ConnectionFactory connectionFactory = (ConnectionFactory)
    initialContext.lookup("ConnectionFactory");
```

The connection factory is used to create a connection, and the connection is used to create a session, as follows:

```
Connection connection = connectionFactory.createConnection("guest",
    "guest");
Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Programs that use JNDI close both the connection and the JNDI context, as follows:

```
connection.close();
getInitialContext().close();
```

## 6.2.2. JNDI Properties for MRG Messaging

MRG Messaging supports the properties shown in the following table:

Property	Purpose
connectionfactory.<jndiname>	The Connection URL that the connection factory will use to perform connections.
queue.<jndiname>	A JMS queue, which is implemented as an amq.direct exchange in MRG Messaging.
topic.<jndiname>	A JMS topic, which is implemented as an amq.topic exchange in MRG Messaging.
destination.<jndiname>	Can be used for defining all amq destinations, queues, topics and header matching, using a Binding URL (see next table).

Table 6.1. JNDI Properties supported by MRG Messaging

These properties can be serialized to or loaded from a JNDI properties file, such as this one:

```
java.naming.factory.initial =
    org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'

# Register an AMQP destination in JNDI
# destination.[jniName] = [BindingURL]
destination.directQueue = direct://amq.direct//message_queue?
routingkey='routing_key'
```

### 6.2.3. Connection URLs

In JNDI properties, a Connection URL specifies properties for a connection. The format for a Connection URL is:

```
amqp://[<user>:<pass>@][<clientid>]<virtualhost>[?<option>='<value>' [&<option>='<value>']]
```

For instance, the following Connection URL specifies a user name, a password, a client ID, a virtual host ("test"), a broker list with a single broker, and a TCP host with the host name "localhost" using port 5672:

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

MRG Messaging supports the following properties in Connection URLs:

Option	Type	Description
brokerlist	see below	The broker to use for this connection. In the current release, precisely one broker must be specified.
maxprefetch	--	The maximum number of pre-fetched messages per destination.
sync_persistence	false	When true, a sync command is sent after every persistent message to guarantee that it has been received.

Table 6.2. Connection URL Properties

Broker lists are specified using a URL in this format:

```
brokerlist=<transport>://<host>[:<port>]
```

For instance, this is a typical broker list:

```
brokerlist='tcp://localhost:5672'
```

### 6.2.4. Binding URLs

In MRG Messaging JNDI properties, a Binding URL can specify the bindings that define the relationship between a queue and an exchange. For instance, the following Binding URL specifies a binding between a message queue named **message\_queue** and the **amq.direct** exchange, using the routing key **routing\_key**:

```
direct://amq.direct//message_queue?routingkey="routing_key"
```

The format for a Binding URL is:



```
[<Exchange Class>://<Exchange Name>/[<Queue>][?
<option>='<value>' [&<option>='<value>']]
```

MRG Messaging supports the following properties in Binding URLs:

Option	Type	Description
durable	boolean	<b>true</b> if queue is durable, <b>false</b> if transient.
exclusive	boolean	<b>true</b> for a private queue, <b>false</b> for a shared queue.
autodelete	boolean	<b>true</b> if queue is automatically deleted when last subscription finishes.
routingkey	string (see below)	Value to use as the routing key for the destination.
bindingkey	string (see below)	One or more binding keys to be bound to the destination.
clientid	string	Client ID.
subscription	boolean	<b>true</b> if a subscription should be created for this queue, <b>false</b> otherwise.

Table 6.3. Binding URL Properties

A routing key is used when publishing, a binding key is used when binding a queue to an exchange. In JMS, a destination can play both roles, so the same destination can be used for both publishing and consuming. A given binding key may have only one routing key, but it may have multiple binding keys. For backwards compatibility, if no binding key is given and the destination is bound to an exchange, the routing key is used as a binding key.

## 6.3. Creating and Closing Connections and Sessions with AMQP

MRG Messaging can establish connections and sessions for use with Java JMS in two different ways. Many programs use JNDI to set up the initial session, then rely on the Java JMS APIs in their program code, as shown in the previous section. This is also the approach used in the examples in this chapter.

It is also possible to use the Qpid APIs to set up a session explicitly in your program, then use the Java APIs for programming with that session. The following skeleton shows how to do that:

```
Connection connection = new AMQConnection(broker_server, broker_port,
    broker_login, broker_password, "clientid/test", "");

connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

// ----- Your code goes here -----

session.close();
```

```
connection.close();
```

For instance, the following code can be inserted into the above framework to create a queue, send a message, and read it back:

```
Destination destinationQueue = new AMQQueue("amq.direct", "weather");

MessageProducer producerQueue = session.createProducer(destinationQueue);

TextMessage txtQueueMsg = session.createTextMessage();
txtQueueMsg.setText("Hello Queue");

producerQueue.send(txtQueueMsg);

MessageConsumer consumerQueue = session.createConsumer(destinationQueue);

Message msgQueue = consumerQueue.receive();
msgQueue.acknowledge();
```

## 6.4. Writing Direct Applications in Java JMS

This section describes three programs that implement direct messaging in Java JMS:

- **Producer.java** publishes messages to the queue associated with the routing key **routing\_key**.
- **Consumer.java** uses a message consumer to receive messages from the queue associated with the routing key **age\_queue**.
- **Listener.java** uses a message listener to receive messages from the queue associated with the routing key **age\_queue**.

Unlike the C++ and Python examples, our Java JMS examples use JNDI to establish the server environment instead of using a separate configuration program. The publisher and listener each look for a queue with the correct name. The JMS library transparently creates destinations on the broker when required.

### 6.4.1. Running the Direct Examples

The example programs discussed in this section are found in `/usr/share/doc/rhm-0.3/java/`. Copy this directory to a location where you have write privileges. To run these programs, do the following:

1. Make sure that a qpidd broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **java** directory, use **runSample.sh** to run the **Consumer** program:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.direct.Consumer
```

```
Using QPID_HOME: /usr/share/java/  
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3  
Consumer: Setting an ExceptionListener on the connection as sample uses  
a MessageConsumer  
Consumer: Creating a non-transacted, auto-acknowledged session  
Consumer: Creating a MessageConsumer  
Consumer: Starting connection so MessageConsumer can receive messages
```

3. In a separate window, use **runSample.sh** to run the **Producer** program:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.direct.Producer  
Using QPID_HOME: /usr/share/java/  
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3  
Producer: Creating a non-transacted, auto-acknowledged session  
Producer: Creating a Message Producer  
Producer: Creating a TestMessage to send to the destination  
Producer: Sending message: 1  
Producer: Sending message: 2  
Producer: Sending message: 3  
Producer: Sending message: 4  
Producer: Sending message: 5  
Producer: Sending message: 6  
Producer: Sending message: 7  
Producer: Sending message: 8  
Producer: Sending message: 9  
Producer: Sending message: 10  
Producer: Closing connection  
Producer: Closing JNDI context
```

4. Now go back to the window where the **Consumer** program is running. You should see the following output:

```
Consumer: Received message: Message 1  
Consumer: Received message: Message 2  
Consumer: Received message: Message 3  
Consumer: Received message: Message 4  
Consumer: Received message: Message 5  
Consumer: Received message: Message 6  
Consumer: Received message: Message 7  
Consumer: Received message: Message 8  
Consumer: Received message: Message 9  
Consumer: Received message: Message 10  
Consumer: Received final message That's all, folks!  
Consumer: Closing connection  
Consumer: Closing JNDI context
```

### 6.4.2. JNDI Properties

The examples in this section all use the following JNDI properties. The connection factory URL includes the password, user name, and host address. The destination declares a queue bound to the routing key **routing\_key**.

```
java.naming.factory.initial =
  org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'

# Register an AMQP destination in JNDI
# destination.[jniName] = [BindingURL]
destination.directQueue = direct://amq.direct//message_queue?
routingkey='routing_key'
```

### 6.4.3. Publishing Messages to a Queue

This section describes **Producer.java**, which sends messages to the direct exchange. The producer class is derived from **BaseExample**, which gives it access to the JNDI environment described in the previous section.

Before we can publish messages, we must have an open connection, a session, and a destination queue. First we get a connection factory using JNDI:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("direct.properties"));

//Create the initial context
Context ctx = new InitialContext(properties);

// look up destination and connection factory
Destination destination = (Destination)ctx.lookup("directQueue");
ConnectionFactory conFac =
  (ConnectionFactory)ctx.lookup("qpidConnectionFactory");
```

Create the connection and the session using this connection factory:

```
Connection connection = conFac.createConnection();
Session session = connection.createSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

Create a **MessageProducer** and a message, using the Message object to send a series of messages in a loop.



### Note

In Java JMS, we do not need to start the connection in order to send a message. We do need to start a connection in order to receive a message.

```
MessageProducer messageProducer = session.createProducer(destination);
TextMessage message;

// Send a series of messages in a loop
for (int i = 1; i < getNumberMessages() + 1; i++)
{
    message = session.createTextMessage("Message " + i);
    messageProducer.send(message, getDeliveryMode(),
        Message.DEFAULT_PRIORITY, Message.DEFAULT_TIME_TO_LIVE);
}
```

After sending the messages, send a final termination message:

```
message = session.createTextMessage("That's all, folks!");
messageProducer.send(message, getDeliveryMode(), Message.DEFAULT_PRIORITY,
    Message.DEFAULT_TIME_TO_LIVE);
```

Now we need only close the connection and the JNDI context.

```
connection.close();
getInitialContext().close();
```

## 6.4.4. Reading Messages from the Queue with a Message Consumer

In this section we will see how to read messages from the queue using a Message Consumer. The code used in this section is taken from **Consumer.java**.

The first step is to get a connection and create a session. We also create an exception handler for the connection:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("direct.properties"));

//Create the initial context
Context ctx = new InitialContext(properties);

// look up destination and connection factory
Destination destination = (Destination)ctx.lookup("directQueue");
ConnectionFactory conFac =
    (ConnectionFactory)ctx.lookup("qpidConnectionFactory");

Connection connection = conFac.createConnection();
```

```
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        System.err.println(CLASS + ": The sample received an exception
through the ExceptionListener");
        System.exit(0);
    }
});

Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Message consumers need to start the connection in order to receive messages. Create a `messageConsumer` and start the connection:

```
MessageConsumer messageConsumer = session.createConsumer(destination);
connection.start();
```

Now we can read our messages and print them out, terminating when the final message is received:

```
Message message;
boolean end = false;
while (!end)
{
    message = messageConsumer.receive();
    String text;
    if (message instanceof TextMessage)
    {
        text = ((TextMessage) message).getText();
    }
    else
    {
        byte[] body = new byte[(int) ((BytesMessage)
message).getBodyLength()];
        ((BytesMessage) message).readBytes(body);
        text = new String(body);
    }
    if (text.equals("That's all, folks!"))
    {
        System.out.println(CLASS + ": Received final message " + text);
        end = true;
    }
    else
    {
        System.out.println(CLASS + ": Received message: " + text);
    }
}
```

Once we have received all messages, we close the connection and the JNDI context before terminating:

```
connection.close();
getInitialContext().close();
```

### 6.4.5. Reading Messages from the Queue using a Message Listener

In the previous section, we read messages using a message consumer. In this section we read using a message listener, which receives messages asynchronously. We create the connection, session, and queue precisely as for the message consumer:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("direct.properties"));

//Create the initial context
Context ctx = new InitialContext(properties);

// look up destination and connection factory
Destination destination = (Destination)ctx.lookup("directQueue");
ConnectionFactory conFac =
    (ConnectionFactory)ctx.lookup("qpidConnectionFactory");

Connection connection = conFac.createConnection();
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        System.err.println(CLASS + ": The sample received an exception
through the ExceptionListener");
        System.exit(0);
    }
});

Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

The class we are implementing is the message listener class, we register it with the message consumer object, then wait for messages to arrive. We use the flag **\_failed** to indicate that an invalid message has been received.

```
MessageConsumer messageConsumer = session.createConsumer(destination);

// Set the listener and start the connection
messageConsumer.setMessageListener(this);
connection.start();

// Wait for the messageConsumer to have received all the messages it needs
synchronized (_lock)
{
    while (!_finished && !_failed)
    {
```

```
        _lock.wait();
    }
}
if (_failed)
{
    System.out.println(CLASS + ": ERROR: invalid message(s)");
}

// Once all messages have been received, close the connection and the JNDI
// context
connection.close();
getInitialContext().close();
```

A message listener implements an **onMessage()** method, which takes a **Message** as a parameter. Our **onMessage** method prints the message, checks to see if it is the final message, and signals termination if it is. If an exception is received, we set **\_failed** to true before termination.

```
public void onMessage(Message message)
{
    try
    {
        String text;
        if (message instanceof TextMessage)
        {
            text = ((TextMessage) message).getText();
        }
        else
        {
            byte[] body = new byte[(int) ((BytesMessage)
message).getBodyLength()];
            ((BytesMessage) message).readBytes(body);
            text = new String(body);
        }
        if (text.equals("That's all, folks!"))
        {
            System.out.println(CLASS + ": Received final message " + text);
            synchronized (_lock)
            {
                _finished = true;
                _lock.notifyAll();
            }
        }
        else
        {
            System.out.println(CLASS + ": Received message: " + text);
        }
    }
    catch (JMSEException exp)
    {
        System.out.println(CLASS + ": Caught an exception handling a
received message");
    }
}
```



```

        exp.printStackTrace();
        synchronized (_lock)
        {
            _failed = true;
            _lock.notifyAll();
        }
    }
}

```

## 6.5. Writing Fanout Applications in Java JMS

This section describes three programs that implement fanout messaging in Java JMS:

- **Producer.java** publishes messages to the **amq.fanoute** exchange.
- **Consumer.java** uses a message consumer to receive messages from the **amq.fanout** exchange.
- **Listener.java** uses a message listener to receive messages from the queue named **message\_queue**.

Unlike the C++ and Python examples, our Java JMS examples use JNDI to establish the server environment instead of using a separate configuration program. In this example, every message that the producer writes to the fanout exchange is sent to the queues used by each listener or consumer.

### 6.5.1. Running the Fanout Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/java/**. Copy this directory to a location where you have write privileges. To run these programs, do the following:

1. Make sure that a **qpidd** broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **java** directory, use **runSample.sh** to run the **Consumer** or **Listener** program, specifying a unique queue name, which must be "fanoutQueue1", "fanoutQueue2", or "fanoutQueue3":

```

$ ./runSample.sh org.apache.qpid.example.jmsexample.fanout.Consumer
fanoutQueue1
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Consumer: Setting an ExceptionListener on the connection as sample uses
a MessageConsumer
Consumer: Creating a non-transacted, auto-acknowledged session
Consumer: Creating a MessageConsumer
Consumer: Starting connection so MessageConsumer can receive messages

```

You can do this in up to three windows, specifying a different name for each queue.

3. In a separate window, use **runSample.sh** to run the **Producer** program:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.fanout.Producer
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Producer: Creating a non-transacted, auto-acknowledged session
Producer: Creating a Message Producer
Producer: Creating a TestMessage to send to the destination
Producer: Sending message: 1
Producer: Sending message: 2
Producer: Sending message: 3
Producer: Sending message: 4
Producer: Sending message: 5
Producer: Sending message: 6
Producer: Sending message: 7
Producer: Sending message: 8
Producer: Sending message: 9
Producer: Sending message: 10
Producer: Closing connection
Producer: Closing JNDI context
```

4. Now go back to the window where the **Listener** program is running. You should see output like this:

```
Consumer: Received message: Message 1
Consumer: Received message: Message 2
Consumer: Received message: Message 3
Consumer: Received message: Message 4
Consumer: Received message: Message 5
Consumer: Received message: Message 6
Consumer: Received message: Message 7
Consumer: Received message: Message 8
Consumer: Received message: Message 9
Consumer: Received message: Message 10
Consumer: Received final message That's all, folks!
Consumer: Closing connection
Consumer: Closing JNDI context
```

### 6.5.2. JNDI Properties

The fanout exchange routes every message to every queue that is bound to it. This application uses the following JNDI properties file:

```
java.naming.factory.initial =
  org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
```

```
# connectionfactory.[jndiname] = [ConnectionFactory]
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'

# Register an AMQP destination in JNDI
# destination.[jniName] = [BindingURL]
destination.fanoutQueue1 = fanout://amq.fanout//message_queue1
destination.fanoutQueue2 = fanout://amq.fanout//message_queue2
destination.fanoutQueue3 = fanout://amq.fanout//message_queue3

# for producer
destination.fanoutQueue = fanout://amq.fanout//message_queue
```

### 6.5.3. Reading Messages from a Queue with a Message Consumer

In this section we will see how to read messages from a queue using a Message Consumer. The code used in this section is taken from **Consumer.java**. As you will see, this program is extremely similar to the consumer used in for the direct example, differing only in the JNDI property file used and the code used to ensure that each consumer has a queue with a unique name.

The first step is to get a connection and create a session. We also create an exception handler for the connection:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("fanout.properties"));

//Create the initial context
Context ctx = new InitialContext(properties);

// look up destination and connection factory
Destination destination = (Destination)ctx.lookup(queueName);
ConnectionFactory conFac =
    (ConnectionFactory)ctx.lookup("qpidConnectionFactory");

Connection connection = conFac.createConnection();
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        System.err.println(CLASS + ": The sample received an exception
        through the ExceptionListener");
        System.exit(0);
    }
});

System.out.println(CLASS + ": Creating a non-transacted, auto-acknowledged
    session");
Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Message consumers need to start the connection in order to receive messages. Create a `messageConsumer` and start the connection:

```
MessageConsumer messageConsumer = session.createConsumer(destination);
connection.start();
```

Now we can read our messages and print them out, terminating when the final message is received:

```
Message message;
boolean end = false;
while (!end)
{
    message = messageConsumer.receive();
    String text;
    if (message instanceof TextMessage)
    {
        text = ((TextMessage) message).getText();
    }
    else
    {
        byte[] body = new byte[(int) ((BytesMessage)
message).getBodyLength()];
        ((BytesMessage) message).readBytes(body);
        text = new String(body);
    }
    if (text.equals("That's all, folks!"))
    {
        System.out.println(CLASS + ": Received final message " + text);
        end = true;
    }
    else
    {
        System.out.println(CLASS + ": Received message: " + text);
    }
}
```

Once we have received all messages, we close the connection and the JNDI context before terminating:

```
connection.close();
ctx.close();
```

### 6.5.4. Reading Messages from the Queue using a Message Listener

In the previous section, we read messages using a message consumer. In this section we read using a message listener, which receives messages asynchronously. As you will see, this program is extremely similar to the listener used in for the direct example, differing only in the JNDI property file used and the code used to ensure that each listener has a queue with a unique name. We create the connection, session, and queue precisely as for the message consumer:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("fanout.properties"));

// Create the initial context
Context ctx = new InitialContext(properties);

// Look up destination and connection factory
Destination destination = (Destination)ctx.lookup(queueName);
ConnectionFactory conFac =
    (ConnectionFactory)ctx.lookup("qpidConnectionFactory");

Connection connection = conFac.createConnection();
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        // The connection may have broken invoke reconnect code if
        // available.
        System.err.println(CLASS + ": The sample received an exception
        through the ExceptionListener");
        System.exit(0);
    }
});

Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

The class we are implementing is the message listener class, we register it with the message consumer object, then wait for messages to arrive. We use the flag **\_failed** to indicate that an invalid message has been received.

```
MessageConsumer messageConsumer = session.createConsumer(destination);

// Set the listener and start the connection
messageConsumer.setMessageListener(this);
connection.start();

// Wait for the messageConsumer to have received all the messages it needs
// synchronized (_lock)
{
    while (!_finished && !_failed)
    {
        _lock.wait();
    }
}
if (_failed)
{
    System.out.println(CLASS + ": This sample failed as it received
    unexpected messages");
}
```

```
// Once all messages have been received, close the connection and the JNDI
context
connection.close();
ctx.close();
```

A message listener implements an `onMessage()` method, which takes a `Message` as a parameter. Our `onMessage` method prints the message, checks to see if it is the final message, and signals termination if it is. If an exception is received, we set `_failed` to true before termination.

```
public void onMessage(Message message)
{
    try
    {
        String text;
        if (message instanceof TextMessage)
        {
            text = ((TextMessage) message).getText();
        }
        else
        {
            byte[] body = new byte[(int) ((BytesMessage)
message).getBodyLength()];
            ((BytesMessage) message).readBytes(body);
            text = new String(body);
        }
        if (text.equals("That's all, folks!"))
        {
            System.out.println(CLASS + ": Received final message " + text);
            synchronized (_lock)
            {
                _finished = true;
                _lock.notifyAll();
            }
        }
        else
        {
            System.out.println(CLASS + ": Received message: " + text);
        }
    }
    catch (JMSEException exp)
    {
        System.out.println(CLASS + ": Caught an exception handling a
received message");
        exp.printStackTrace();
        synchronized (_lock)
        {
            _failed = true;
            _lock.notifyAll();
        }
    }
}
```

```
}
```

### 6.5.5. Publishing Messages to a Fanout Exchange

This section describes **Producer.java**, which sends messages to a fanout exchange. The producer class is derived from **BaseExample**, which gives it access to the JNDI environment described in the previous section.

Before we can publish messages, we must have an open connection, a session, and a destination queue. First we get a connection factory using JNDI:

```
// Load JNDI properties
Properties properties = new Properties();
properties.load(this.getClass().getResourceAsStream("fanout.properties"));

// Create the initial context
Context ctx = new InitialContext(properties);

Destination destination = (Destination)ctx.lookup("fanoutQueue");

// Declare the connection
ConnectionFactory conFac =
    (ConnectionFactory)ctx.lookup("qpidConnectionFactory");
```

Create the connection and the session using this connection factory:

```
Connection connection = conFac.createConnection();
Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Create a **MessageProducer** and a message, using the **Message** object to send a series of messages in a loop.



#### Note

In Java JMS, we do not need to start the connection in order to send a message. We do need to start a connection in order to receive a message.

```
MessageProducer messageProducer = session.createProducer(destination);
TextMessage message;

// Send a series of messages in a loop
for (int i = 1; i < numMessages + 1; i++)
{
    message = session.createTextMessage("Message " + i);
    messageProducer.send(message, deliveryMode, Message.DEFAULT_PRIORITY,
        Message.DEFAULT_TIME_TO_LIVE);
}
```

After sending the messages, send a final termination message:

```
message = session.createTextMessage("That's all, folks!");
messageProducer.send(message, deliveryMode, Message.DEFAULT_PRIORITY,
    Message.DEFAULT_TIME_TO_LIVE);
```

Now we need only close the connection and the JNDI context.

```
connection.close();
ctx.close();
```

## 6.6. Writing Publish/Subscribe Applications in Java JMS

This section describes two programs that implement Publish/Subscribe messaging:

- **Publisher.java** sends messages to the hierarchical topics **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**.
- **Listener.java** listens for messages sent to the topics **#.news**, **#.weather**, **europe.#**, and **usa.#**.

In this example, the publisher creates messages for topics like news, weather, and sports that happen in regions like Europe, Asia, or the United States. A given consumer may be interested in all weather messages, regardless of region, or it may be interested in news and weather for the United States, but uninterested in items for other regions. In this example, each consumer sets up its own private queues, which receive precisely the messages that particular consumer is interested in.

### 6.6.1. Running the Publish/Subscribe Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/java/**. Copy this directory to a location where you have write privileges. To run these programs, do the following:

1. Make sure that a qpidd broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the **qpidd** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **java** directory, use **runSample.sh** to run the **Listener** program:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.pubsub.Listener
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Listener: Setting an ExceptionListener on the connection as sample uses
a TopicSubscriber
Listener: Creating a non-transacted, auto-acknowledged session
Listener: Creating a Message Subscriber for topic usa
Listener: Creating a Message Subscriber for topic europe
Listener: Creating a Message Subscriber for topic news
Listener: Creating a Message Subscriber for topic weather
```



```
Listener: Starting connection so TopicSubscriber can receive messages
```

3. In a separate window, use **runSample.sh** to run the **Publisher** program:

```
$ ./runSample.sh org.apache.qpid.example.jmsexample.pubsub.Publisher
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Publisher: Creating a non-transacted, auto-acknowledged session
Publisher: Creating a TestMessage to send to the topics
Publisher: Creating a Message Publisher for topic usa.weather
Publisher: Sending message 1
Publisher: Sending message 2
Publisher: Sending message 3
Publisher: Sending message 4
Publisher: Sending message 5
Publisher: Sending message 6
Publisher: Creating a Message Publisher for topic usa.news
Publisher: Sending message 1
Publisher: Sending message 2
Publisher: Sending message 3
Publisher: Sending message 4
Publisher: Sending message 5
Publisher: Sending message 6
Publisher: Creating a Message Publisher for topic europe.weather
Publisher: Sending message 1
Publisher: Sending message 2
Publisher: Sending message 3
Publisher: Sending message 4
Publisher: Sending message 5
Publisher: Sending message 6
Publisher: Creating a Message Publisher for topic europe.news
Publisher: Sending message 1
Publisher: Sending message 2
Publisher: Sending message 3
Publisher: Sending message 4
Publisher: Sending message 5
Publisher: Sending message 6
Publisher: Closing connection
Publisher: Closing JNDI context
```

4. Now go back to the window where the **Listener** program is running. You should see output like this:

```
Listener: Received message for topic: usa: message 1
Listener: Received message for topic: weather: message 1
Listener: Received message for topic: usa: message 2
Listener: Received message for topic: weather: message 2
Listener: Received message for topic: usa: message 3
Listener: Received message for topic: weather: message 3
Listener: Received message for topic: usa: message 4
```

```
Listener: Received message for topic: weather: message 4
Listener: Received message for topic: usa: message 5
Listener: Received message for topic: weather: message 5
Listener: Received message for topic: usa: message 6
Listener: Received message for topic: weather: message 6
. . .
Listener: Shutting down listener for news
Listener: Shutting down listener for weather
Listener: Shutting down listener for usa
Listener: Shutting down listener for europe
Listener: Closing connection
Listener: Closing JNDI context
```

### 6.6.2. JNDI Properties

From a Java JMS perspective, we will be using topics, which are implemented using an AMQP topic exchange in MRG Messaging. The following properties file creates the topics we need.

```
java.naming.factory.initial =
  org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'

# register some topics in JNDI using the form
# topic.[jndiName] = [physicalName]
topic.usa.weather = usa.weather,control
topic.usa.news = usa.news,control
topic.europe.weather = europe.weather,control
topic.europe.news = europe.news,control
topic.weather = #.weather,control
topic.news = #.news,control
topic.europe = europe.#,control
topic.usa = usa.#,control
topic.control = control
```

In the above JNDI file, note that we can map more than one key to a given topic. For instance, **topic.usa.weather = usa.weather,control** places both messages with the routing key **usa.weather** and the routing key **control** on **topic.usa.weather**.

### 6.6.3. Publishing Messages to a Topic

**Publisher.java** starts by creating a connection and a session, then creating a **TextMessage** object from the session. We will also create an exception listener to handle any JMS exceptions we receive:

```
Properties properties=new Properties();
properties.load(this.getClass().getResourceAsStream("pubsub.properties"));
```

```
//Create the initial context
Context ctx=new InitialContext(properties);

// Declare the connection factory, create a connection and a session
ConnectionFactory conFac=(ConnectionFactory)
    ctx.lookup("qpidConnectionFactory");
TopicConnection connection= (TopicConnection) conFac.createConnection();
TopicSession session=connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Now we create a text message from the session:

```
message=session.createTextMessage();
```

To publish a message to a particular topic, we first look up the topic, then create a TopicPublisher and publish messages using it:

```
Topic topic = (Topic)ctx.lookup("usa.weather");

TopicPublisher messagePublisher=session.createPublisher(topic);
publishMessages(message, messagePublisher);
```

The publishMessages() method simply publishes a series of messages:

```
private void publishMessages(TextMessage message, TopicPublisher
    messagePublisher) throws JMSEException
{
    for (int i = 1; i < getNumberMessages() + 1; i++)
    {
        message.setText("Message " + i);
        messagePublisher
            .send(message, getDeliveryMode(), Message.DEFAULT_PRIORITY,
                Message.DEFAULT_TIME_TO_LIVE);
    }
}
```

Now we send one final message to indicate termination:

```
// send the final message
message=session.createTextMessage("That's all, folks!");
topic = (Topic)ctx.lookup("control");
// Create a Message Publisher
messagePublisher = session.createPublisher(topic);
messagePublisher
    .send(message, DeliveryMode.PERSISTENT, Message.DEFAULT_PRIORITY,
        Message.DEFAULT_TIME_TO_LIVE);
```

And close the connection and the JNDI context:

```
connection.close();
getInitialContext().close();
```

### 6.6.4. Reading Messages from the Queue

The second program in the publish/subscribe example, **Listener.java** listens for messages from the wildcard topics **usa.#**, **europe.#**, **#.news**, and **#.weather**.

We start by creating a connection and a session as in the other examples:

```
Properties properties=new Properties();
properties.load(this.getClass().getResourceAsStream("pubsub.properties"));

Context ctx=new InitialContext(properties);

ConnectionFactory conFac=(ConnectionFactory)
    ctx.lookup("qpidConnectionFactory");
TopicConnection connection=(TopicConnection) conFac.createConnection();

connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        System.err.println(CLASS + ": The sample received an exception
            through the ExceptionListener");
        System.exit(0);
    }
});

TopicSession session=connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

We write a function to create a message subscriber for a given topic:

```
private void createListener(Context ctx,TopicSession session,String
    topicName) throws Exception{
    // lookup the topic usa
    Topic topic=(Topic) ctx.lookup(topicName);
    // Create a Message Subscriber
    System.out.println(CLASS + ": Creating a Message Subscriber for topic "
+ topicName);
    javax.jms.TopicSubscriber
    messageSubscriber=session.createSubscriber(topic);

    // Set a message listener on the messageConsumer
    messageSubscriber.setMessageListener(new MyMessageListener(topicName));
}
```

Now we create a session for the connection,create listeners for each topic that interests us, and start the connection:

```

System.out.println(CLASS + ": Creating a non-transacted, auto-acknowledged
session");
TopicSession session=connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

createListener(ctx,session,"usa");
createListener(ctx,session,"europe");
createListener(ctx,session,"news");
createListener(ctx,session,"weather");

```

Now we start the connection to begin receiving messages, and wait until all messages have been received:

```

connection.start();

// Wait for the messageConsumer to have received all the messages it needs
synchronized (_lock)
{
    while (_finished < 4 && !_failed)
    {
        _lock.wait();
    }
}

// If the MessageListener abruptly failed (probably due to receiving a non-
text message)
if (_failed)
{
    System.out.println(CLASS + ": This sample failed as it received
unexpected messages");
}

```

Once all messages have been received, we close the connection and the JNDI context:

```

connection.close();
getInitialContext().close();

```

Here is the code for the MessageListener used in this example:

```

private class MyMessageListener implements MessageListener
{
    /* The topic this subscriber is subscribing to */
    private String _topicName;

    public MyMessageListener(String topicName)
    {
        _topicName=topicName;
    }

    public void onMessage(Message message)

```

```
{
    try
    {
        String text;
        if (message instanceof TextMessage)
        {
            text=((TextMessage) message).getText();
        }
        else
        {
            byte[] body=new byte[(int) ((BytesMessage)
message).getBodyLength()];
            ((BytesMessage) message).readBytes(body);
            text=new String(body);
        }
        if (text.equals("That's all, folks!"))
        {
            System.out.println(CLASS + ": Shutting down listener
for " + _topicName);
            synchronized (_lock)
            {
                _finished++;
                _lock.notifyAll();
            }
        }
        else
        {
            System.out.println(CLASS + ": Received message for
topic: " + _topicName + ": " + text);
        }
    }
    catch (JMSEException exp)
    {
        System.out.println(CLASS + ": Caught an exception handling
a received message");
        exp.printStackTrace();
        synchronized (_lock)
        {
            _failed=true;
            _lock.notifyAll();
        }
    }
}
}
```

## 6.7. Writing Request/Response Applications in Java JMS

In the request/response examples, we write a server that accepts strings from clients and converts them to upper case, sending the result back to the requesting client. This example consists of two programs.

- **Client.java** is a client application that sends lines of poetry to the server.
- **Server.java** is a simple service that accepts messages, converts their content to upper case, and sends the result to the **amq.direct** exchange, using the request's **reply-to** property as the routing key for the response.

### 6.7.1. JNDI Properties

The JNDI properties for this example simply create a request queue.

```
java.naming.factory.initial =
  org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/test?
brokerlist='tcp://localhost:5672'

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
queue.requestQueue = request
```

### 6.7.2. Running the Request/Response Examples

The example programs discussed in this section are found in **/usr/share/doc/rhm-0.3/java/**. Copy this directory to a location where you have write privileges. To run these programs, do the following:

1. Make sure that a qpid broker is running:

```
$ ps -eaf | grep qpid
```

If a broker is running, you should see the **qpid** process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. In the **java** directory, use **runSample.sh** to run the **Server** program:

```
$ ./runSample.sh
  org.apache.qpid.example.jmsexample.requestResponse.Server
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Server: Setting an ExceptionListener on the connection as sample uses a
  MessageConsumer
Server: Creating a non-transacted, auto-acknowledged session
Server: Creating a MessageConsumer
Server: Creating a MessageProducer
Server: Starting connection so MessageConsumer can receive messages
```

3. In a separate window, use **runSample.sh** to run the **Client** program:

```
$ ./runSample.sh
org.apache.qpid.example.jmsexample.requestResponse.Client
Using QPID_HOME: /usr/share/java/
Using QPID_SAMPLE: /usr/share/doc/rhm-0.3
Client: Setting an ExceptionListener on the connection as sample uses a
MessageConsumer
Client: Creating a non-transacted, auto-acknowledged session
Client: Creating a QueueRequestor
Client: Starting connection
Client:      Request Content= Twas brillig, and the slithy toves
Client:      Response Content= TWAS BRILLIG, AND THE SLITHY TOVES
Client:      Request Content= Did gire and gymble in the wabe.
Client:      Response Content= DID GIRE AND GYMBLE IN THE WABE.
Client:      Request Content= All mimsy were the borogroves,
Client:      Response Content= ALL MIMSY WERE THE BOROGROVES,
Client:      Request Content= And the mome raths outgrabe.
Client:      Response Content= AND THE MOME RATHS OUTGRABE.
Client: Closing connection
Client: Closing JNDI context
```

### 6.7.3. Client

**Client.java** is the client application. It starts by creating a connection, a session, and a destination queue, to which requests are sent. It also creates an exception listener for the connection so that any message errors can be caught.

```
// Load JNDI properties
Properties properties=new Properties();
properties.load(this.getClass().getResourceAsStream("requestResponse.properties"));

//Create the initial context
Context ctx=new InitialContext(properties);

// Lookup the connection factory
ConnectionFactory conFac = (ConnectionFactory)
    ctx.lookup("qpidConnectionFactory");

// create the connection
QueueConnection connection = (QueueConnection) conFac.createConnection();
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        // The connection may have broken invoke reconnect code if
        // available.
        System.err.println(CLASS + ": The sample received an exception
        through the ExceptionListener");
        System.exit(0);
    }
}
```



```
});
```

The session for this example is a Java JMS `QueueSession`.

```
QueueSession session = connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

**Client** uses a Java JMS **QueueRequestor**, a class that directly supports the kind of service request that is at the heart of this example. We will use the request queue as our destination.

```
Queue destination = (Queue) ctx.lookup("requestQueue");
QueueRequestor requestor = new QueueRequestor(session, destination);
```

Now we start the connection so that we can receive responses to our requests.

```
connection.start();
```

We will use a function called **sendReceive()** to send a request and receive the response. Here is the code that calls **sendReceive()**, which is described later.

```
TextMessage request;

String[] messages = {"Twas brillig, and the slithy toves",
    "Did gire and gymble in the wabe.",
    "All mimsy were the borogroves,",
    "And the mome raths outgrabe."};

for (String message : messages)
{
    request = session.createTextMessage(message);
    sendReceive(request, requestor);
}
```

After calling **sendReceive()**, we close down as in the other examples.

```
connection.close();
getInitialContext().close();
```

The **sendReceive()** function simply sends a request, receives the response, and prints it:

```
private void sendReceive(TextMessage request, QueueRequestor requestor)
    throws JMSEException
{
    Message response;
    response=requestor.request(request);
    System.out.println(CLASS + ": \tRequest Content= " +
        request.getText());
}
```

```
String text;
if (response instanceof TextMessage)
{
    text=((TextMessage) response).getText();
}
else
{
    byte[] body=new byte[(int) ((BytesMessage)
response).getBodyLength()];
    ((BytesMessage) response).readBytes(body);
    text=new String(body);
}
System.out.println(CLASS + ": \tResponse Content= " + text);
}
```

### 6.7.4. The Server

**Server.java** is a server that converts the text of requests to upper case and returns the result to the original sender. It starts by creating a connection and a session:

```
// Load JNDI properties
Properties properties=new Properties();
properties.load(this.getClass().getResourceAsStream("requestResponse.properties"));

//Create the initial context
Context ctx=new InitialContext(properties);

// Lookup the connection factory
ConnectionFactory conFac = (ConnectionFactory)
    ctx.lookup("qpidConnectionFactory");

// create the connection
QueueConnection connection = (QueueConnection) conFac.createConnection();
connection.setExceptionListener(new ExceptionListener()
{
    public void onException(JMSEException jmse)
    {
        // The connection may have broken invoke reconnect code if
        // available.
        System.err.println(CLASS + ": The sample received an exception
        through the ExceptionListener");
        System.exit(0);
    }
});

Session session=connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Next, it creates a Consumer to receive requests from the request queue, and a Producer to send responses:

```
Queue destination = (Queue) ctx.lookup("requestQueue");
```

```
MessageConsumer messageConsumer = session.createConsumer(destination);
MessageProducer messageProducer;
```

Now we start the connection so we can receive requests:

```
connection.start();
```

When a message is received, it is checked to see if it is a text message and has a **ReplyTo** field; if it does, the request is converted to upper case and the response is sent:

```
Message requestMessage;
TextMessage responseMessage;
boolean end=false;
while (!end)
{
    requestMessage=messageConsumer.receive();

    String text;
    if (requestMessage instanceof TextMessage)
    {
        text=((TextMessage) requestMessage).getText();
    }
    else
    {
        byte[] body=new byte[(int) ((BytesMessage)
requestMessage).getBodyLength()];
        ((BytesMessage) requestMessage).readBytes(body);
        text=new String(body);
    }

    if (requestMessage.getJMSReplyTo() != null)
    {
        responseMessage=session.createTextMessage();
        responseMessage.setText(text.toUpperCase());

        messageProducer=session.createProducer(requestMessage.getJMSReplyTo());
        messageProducer.send(responseMessage);
    }
}
```

Then we close the connection and the JNDI context:

```
connection.close();
getInitialContext().close();
```

## 6.8. Durability and Persistence in Java JMS

The Java JMS model of durability and persistence is somewhat different from the native AMQP model, but MRG Messaging supports the Java JMS model natively on top of AMQP.

In particular, we support durable subscribers, and messages are persistent by default. Persistence can be disabled using standard Java JMS mechanisms.

### 6.9. Using Transactions in Java JMS

MRG Messaging supports standard Java JMS transacted sessions. When a session is created it can be made transactional by setting the first parameter of `createSession()` to `true`:

```
Session transactedSession = connection.createSession(true,  
    Session.SESSION_TRANSACTED);
```

Producers and consumers that are created by a transacted session are transactional, and are governed by commits and rollbacks made to that session:

```
MessageConsumer transactedConsumer =  
    transactedSession.createConsumer(queue);  
MessageProducer transactedProducer =  
    transactedSession.createProducer(topic);
```

For example, in the following code, either the messages received and sent are both committed, or they are both rolled back:

```
receivedMessage = transactedConsumer.receive();  
transactedProducer.send(receivedMessage);  
  
if (_commit)  
{  
    transactedSession.commit();  
}  
else  
{  
    transactedSession.rollback();  
}
```

### 6.10. Logging in Java clients

The Java client software supports logging using the Simple Logging Facade for Java (SLF4J), a facade that supports popular logging systems like log4j version 1.2.x, JDK 1.4 logging, Simple logging, and NOP.

This section discusses logging using log4j. Make sure your **CLASSPATH** contains **log4j.1.2.x.jar**, e.g. by installing it with **yum** or by setting the variable by hand..

Log4j requires an initialization file, whose location should be specified as a URL using the **log4j.configuration** system property, e.g:

```
-Dlog4j.configuration=file://home/fidget/java/log4j.xml
```

Let's look at **log4j.xml**, a log4j configuration file shipped in the java examples directory. This property file defines two output destinations, which log4j calls "appenders". One of these writes to a file called "qpid\_messaging.log", the other writes to standard output.

```

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="FileAppender" class="org.apache.log4j.FileAppender">
    <param name="File" value="qpid_messaging.log"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%t %-5p %c{2} - %m%
n"/>
    </layout>
  </appender>

  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%
F:%L) - %m%n"/>
    </layout>
  </appender>
<!-- continued below .... -->

```

This file also uses a separate logger for warning messages in the package **org.apache**, printing messages at level **warn** and above. (log4j uses the following hierarchical set of logging levels: trace, debug, info, warn, error and fatal.)

```

<logger name="org.apache">
  <!-- Print only messages of level warn or above in the package
org.apache -->
  <level value="warn"/>
</logger>
<!-- continued below .... -->

```

In log4j, each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy. The root appender is at the root of the hierarchy, so the following declaration ensures that all logging requests are printed to **stdout**:

```

<root>
  <priority value="info"/>
  <appender-ref ref="STDOUT"/>
  <!-- <appender-ref ref="FileAppender"/> -->
</root>
</log4j:configuration>

```

To ensure that all logging requests are also sent to the **FileAppender**, simply uncomment that element.

More log4j documentation can be found at <http://logging.apache.org/log4j/1.2/manual.html>



# Using MRG Messaging with .NET

This chapter shows how to write direct, fanout, publish/subscribe, and request/response programs in C# using the Apache Qpid .NET 0.10 API. These concepts are explained in [Chapter 2, Examples Overview](#). It then shows how to use important features like persistence and transactions with MRG Messaging. This chapter does not try to teach the entire MRG Messaging C# API, and it is not encyclopedic in its coverage of AMQP. For more detailed information on the AMQP model, see the AMQP specification at <http://www.amqp.org>.

Before running the examples, you need to unzip the file Qpid.NET-net-2.0-M4.zip, which creates the following tree:

```
<home>
|-qpid
  |-lib (contains the required dlls)
  |-examples
    |- direct
      |-example-direct-Listener.exe
      |-example-direct-Producer.exe
    |- fanout
      |-example-fanout-Listener.exe
      |-example-fanout-Producer.exe
    |- pub-sub
      |-example-pub-sub-Listener.exe
      |-example-pub-sub-Publisher.exe
    |- request-response
      |-example-request-response-Client.exe
      |-example-request-response-Server.exe
```

Make sure your **PATH** contains the directory **<home>/qpid/lib**. The compiled examples are available as **.exe** files and can be run from the command line:

```
$ cd <home>/qpid/examples/examplefolder
$ example-...-exe [hostname] [portnumber]
```

where **[hostname]** is the qpid broker host name (default is **localhost**) and **[portnumber]** is the port number on which the qpid broker is accepting connection (default is **5672**).

## 7.1. Creating and Closing Sessions

All of the examples have been written using the Apache Qpid .NET 0.10 API, which is the C# API for MRG Messaging. The examples use the same skeleton code to initialize the program, create a session, and clean up before exiting:

```
using System;
using System.IO;
using System.Text;
```

```
using System.Threading;
using org.apache.qpid.client;
using org.apache.qpid.transport;

...

private static void Main(string[] args)
{
    string host = args.Length > 0 ? args[0] : "localhost";
    int port = args.Length > 1 ? Convert.ToInt32(args[1]) : 5672;
    Client connection = new Client();
    try
    {
        connection.connect(host, port, "test", "guest", "guest");
        ClientSession session = connection.createSession(50000);

        //----- Main body of program
        -----

        connection.close();
    }
    catch (Exception e)
    {
        Console.WriteLine("Error: \n" + e.StackTrace);
    }
}

...
```

## 7.2. Writing Direct Applications in .NET

This section describes two programs that implement direct messaging using a Direct exchange:

- **org.apache.qpid.example.direct.Producer**, found in **example-direct-producer** publishes messages to the **amq.direct** exchange, using the routing key **routing\_key**.
- **org.apache.qpid.example.direct.Listener**, found in **example-direct-Listener** uses a message listener to receive messages from the queue named **message\_queue**.

### 7.2.1. Running the Direct Examples

To run the Direct examples, do the following:

1. Make sure that a qpid broker is running:

```
$ ps -eaf | grep qpid
```

If a broker is running, you should see the qpid process in the output of the above command. If no broker is running, see the instructions in [Chapter 3, Installing MRG Messaging](#).

2. Start one or more instances of the direct listener. The listener waits for messages to be published, as described in the next step, then reads messages from the queue.



To start the listener, first move to the directory that holds the examples, then start the program:

Then start the program:

- cygwin:

```
$ cd <home>/qpid/examples/direct
$ ./example-direct-Listener.exe [hostname] [portnumber]
```

- mono:

```
$ cd <home>/qpid/examples/direct
$ mono ./example-direct-Listener.exe [hostname] [portnumber]
```

3. Publish a series of messages to the **amq.direct** exchange by changing to the directory and starting the program:

Then start the program:

- cygwin:

```
$ cd <home>/qpid/examples/direct
$ ./example-direct-Producer.exe [hostname] [portnumber]
```

- mono:

```
$ cd <home>/qpid/examples/direct
$ mono ./example-direct-Producer.exe [hostname] [portnumber]
```

This program has no output; the messages are routed to the message queue, as instructed by the binding.

4. Go to the windows where you are running your listener. You should see the following output:

```
Message: Message 0
Message: Message 1
Message: Message 2
Message: Message 3
Message: Message 4
Message: Message 5
Message: Message 6
Message: Message 7
Message: Message 8
Message: Message 9
```

```
Message: That's all, folks!
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 7.2.2. Reading Messages from the Queue

The program `listener.cs` is a message listener that receives messages from a queue named `message_queue`.

First it creates a queue named `message_queue`, then binds it to the `amq.direct` exchange using the binding key `routing_key`.

```
// Create a queue named "message_queue", and route all messages whose
// routing key is "routing_key" to this newly created queue.
session.queueDeclare("message_queue");
session.exchangeBind("message_queue", "amq.direct", "routing_key");
```

The queue created by this program continues to exist after the program exits, and any message whose routing key matches the key specified in the binding will be routed to the corresponding queue by the broker. Note that the queue could have been deleted using the following code:

```
session.queueDelete("message_queue");
```

To create a message listener, create a class derived from `IMessageListener`, and override the `messageTransfer` method, providing the code that should be executed when a message is received.

```
public class MessageListener : IMessageListener
{
    .....
    public void messageTransfer(IMessage m)
    {
        .....
    }
}
```

The main body of the program creates a listener for the subscription; attaches the listener to a message queue; and subscribe to the queue to receive messages from the queue.

```
lock (session)
{
    // Create a listener and subscribe it to the queue named "message_queue"
    IMessageListener listener = new MessageListener(session);
    session.attachMessageListener(listener, "message_queue");

    session.messageSubscribe("message_queue");
    // Receive messages until all messages are received
    Monitor.Wait(session);
}
```

```
}
```

The `MessageListener`'s `messageTransfer()` function is called whenever a message is received. In this example the message is printed and tested to see if it is the final message. Once the final message is received, the messages are acknowledged.

```
BinaryReader reader = new BinaryReader(m.Body, Encoding.UTF8);
byte[] body = new byte[m.Body.Length - m.Body.Position];
reader.Read(body, 0, body.Length);
ASCIIEncoding enc = new ASCIIEncoding();
string message = enc.GetString(body);
Console.WriteLine("Message: " + message);
// Add this message to the list of message to be acknowledged
_range.add(m.Id);
if( message.Equals("That's all, folks!") )
{
    // Acknowledge all the received messages
    _session.messageAccept(_range);
    lock(_session)
    {
        Monitor.Pulse(_session);
    }
}
```

### 7.2.3. Publishing Messages to a Direct Exchange

The second program in the direct example, `Producer.cs`, publishes messages to the `amq.direct` exchange using the routing key `routing_key`.

First, create a message and set a routing key. The same routing key will be used for each message we send, so you only need to set this property once.

```
IMessage message = new Message();
// The routing key is a message property. We will use the same
// routing key for each message, so we'll set this property
// just once. (In most simple cases, there is no need to set
// other message properties.)
message.DeliveryProperties.SetRoutingKey("routing_key");
```

Now send some messages:

```
// Asynchronous transfer sends messages as quickly as
// possible without waiting for confirmation.
for (int i = 0; i < 10; i++)
{
    message.clearData();
    message.appendData(Encoding.UTF8.GetBytes("Message " + i));
}
```

```
session.messageTransfer("amq.direct", message);  
}
```

Send a final synchronous message to indicate termination:

```
// And send a synchronious final message to indicate termination.  
message.clearData();  
message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));  
session.messageTransfer("amq.direct", "routing_key", message);  
session.sync();
```

### 7.3. Writing Fanout Applications

This section describes two programs that illustrate the use of a Fanout exchange.

- **Listener.cs** makes a unique queue private for each instance of the listener, and binds that queue to the fanout exchange. All messages sent to the fanout exchange are delivered to each listener's queue.
- **Producer.cs** publishes messages to the fanout exchange. It does not use a routing key, which is not needed by the fanout exchange.

#### 7.3.1. Running the Fanout Examples

1. Make sure your **PATH** contains the directory **<home>/qpidd/lib**
2. Make sure that a qpidd broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the qpidd process in the output of the above command.

3. In separate windows, start one or more fanout listeners as follows:
  - cygwin

```
$ cd <home>/qpidd/examples/direct  
$ ./example-fanout-Listener.exe [hostname] [portnumber]
```

- mono:

```
$ cd <home>/qpidd/examples/direct  
$ mono ./example-fanout-Listener.exe [hostname] [portnumber]
```

The listener creates a private queue, binds it to the amq.fanout exchange, and waits for messages to arrive on the queue. When the listener starts, you will see the following message:

## Listening

This program is waiting for messages to be published, as described in the next step:

4. In a separate window, publish a series of messages to the **amq.fanout** exchange by running the fanout producer, as follows:

- cygwin:

```
$ cd <home>/qpid/examples/direct
$ ./example-fanout-Producer.exe [hostname] [portnumber]
```

- mono:

```
$ cd <home>/qpid/examples/direct
$ mono ./example-fanout-Producer.exe [hostname] [portnumber]
```

This program has no output; the messages are routed to the message queue, as prescribed by the binding.

5. Go to the windows where you are running listeners. You should see the following output for each listener:

```
Message: Message 0
Message: Message 1
Message: Message 2
Message: Message 3
Message: Message 4
Message: Message 5
Message: Message 6
Message: Message 7
Message: Message 8
Message: Message 9
Message: That's all, folks!
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 7.3.2. Consuming from a Fanout Exchange

The first program in the fanout example, **Listener.cs**, creates a private queue, binds it to the **amq.fanout** exchange, and waits for messages to arrive on the queue, printing them out as they arrive. It uses a Listener that is identical to the one used in the direct example:

```
public class MessageListener : IMessageListener
{
    private readonly ClientSession _session;
    private readonly RangeSet _range = new RangeSet();
    public MessageListener(ClientSession session)
    {
        _session = session;
    }

    public void messageTransfer(IMessage m)
    {
        BinaryReader reader = new BinaryReader(m.Body, Encoding.UTF8);
        byte[] body = new byte[m.Body.Length - m.Body.Position];
        reader.Read(body, 0, body.Length);
        ASCIIEncoding enc = new ASCIIEncoding();
        string message = enc.GetString(body);
        Console.WriteLine("Message: " + message);
        // Add this message to the list of message to be acknowledged
        _range.add(m.Id);
        if (message.Equals("That's all, folks!"))
        {
            // Acknowledge all the received messages
            _session.messageAccept(_range);
            lock (_session)
            {
                Monitor.Pulse(_session);
            }
        }
    }
}
```

The listener creates a private queue to receive its messages and binds it to the fanout exchange:

```
string myQueue = session.Name;
session.queueDeclare(myQueue, Option.EXCLUSIVE, Option.AUTO_DELETE);
session.exchangeBind(myQueue, "amq.fanout", "my-key");
```

Now we create a listener and subscribe it to the queue:

```
lock (session)
{
    Console.WriteLine("Listening");
    // Create a listener and subscribe it to my queue.
    IMessageListener listener = new MessageListener(session);
    session.attachMessageListener(listener, myQueue);
    session.messageSubscribe(myQueue);
    // Receive messages until all messages are received
    Monitor.Wait(session);
}
```

```
}
```

### 7.3.3. Publishing Messages to the Fanout Exchange

The second program in this example, **Producer.cs**, writes messages to the fanout queue. Unlike topic exchanges and direct exchanges, a fanout exchange need not set a routing key.

```
IMessage message = new Message();
// Asynchronous transfer sends messages as quickly as
// possible without waiting for confirmation.
for (int i = 0; i < 10; i++)
{
    message.clearData();
    message.appendData(Encoding.UTF8.GetBytes("Message " + i));
    session.messageTransfer("amq.fanout", message);
}
```

And send a final message to indicate termination, synchronizing after the final transfer.

```
message.clearData();
message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));
session.messageTransfer("amq.fanout", message);
session.sync();
```

### 7.3.4. Writing Publish/Subscribe Applications

This section describes two programs that implement Publish/Subscribe messaging using a topic exchange.

- **Publisher.cs** sends messages to the **amq.topic** exchange, using the multipart routing keys **usa.news**, **usa.weather**, **europe.news**, and **europe.weather**.
- **Listener.cs** creates private queues for “news”, “weather”, “usa”, and “europe”, binding them to the **amq.topic** exchange using bindings that match the corresponding parts of the multipart routing keys.

In this example, the publisher creates messages for topics like news, weather, and sports that happen in regions like Europe, Asia, or the United States. A given consumer may be interested in all weather messages, regardless of region, or it may be interested in news and weather for the United States, but uninterested in items for other regions. In this example, each consumer sets up its own private queues, which receive precisely the messages that particular consumer is interested in.

### 7.3.5. Running the Publish-Subscribe Examples

1. Make sure your **PATH** contains the directory **<home>/qpidd/lib**
2. Make sure that a qpid broker is running:

```
$ ps -eaf | grep qpid
```

If a broker is running, you should see the qpid process in the output of the above command.

3. In separate windows, start one or more topic subscribers as follows:

- cywin:

```
$ cd <home>/qpid/examples/direct  
$ ./example-pub-sub-Listener.exe [hostname] [portnumber]
```

- mono:

```
$ cd <home>/qpid/examples/direct  
$ mono ./example-pub-sub-Listener.exe [hostname] [portnumber]
```

You will see output similar to this:

```
Listening for messages ...  
Declaring queue: usa  
Declaring queue: europe  
Declaring queue: news  
Declaring queue: weather
```

Each topic consumer creates a set of private queues, and binds each queue to the **amq.topic** exchange together with a binding that indicates which messages should be routed to the queue.

4. In another window, start the topic publisher, which publishes messages to the **amq.topic** exchange, as follows:

- cygwin

```
$ cd <home>/qpid/examples/direct  
$ ./example-pub-sub-Producer.exe [hostname] [portnumber]
```

- mono

```
$ cd <home>/qpid/examples/direct  
$ mono ./example-pub-sub-Producer.exe [hostname] [portnumber]
```

This program has no output; the messages are routed to the message queues for each topic\_consumer as specified by the bindings the consumer created.

5. Go back to the window for each topic consumer. You should see output like this:



```

Message: Message 0 from usa
Message: Message 0 from news
Message: Message 0 from weather
Message: Message 1 from usa
Message: Message 1 from news
Message: Message 2 from usa
Message: Message 2 from news
Message: Message 3 from usa
Message: Message 3 from news
Message: Message 4 from usa
Message: Message 4 from news
Message: Message 5 from usa
Message: Message 5 from news
Message: Message 6 from usa
Message: Message 6 from news
Message: Message 7 from usa
Message: Message 7 from news
Message: Message 8 from usa
Message: Message 8 from news
Message: Message 9 from usa
....
Message: That's all, folks! from weather
Shutting down listener for control
Message: That's all, folks! from europe
Shutting down listener for control

```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, “Creating and Closing Sessions”](#).

### 7.3.5.1. Publishing Messages to a Topic Exchange

The first program in the publish/subscribe example, **Publisher.cs**, defines two new functions: one that publishes messages to the topic exchange, and one that indicates that no more messages are coming.

The **publishMessages** function publishes a series of five messages using the specified routing key.

```

private static void publishMessages(ClientSession session, string
routing_key)
{
    IMessage message = new Message();
    // Asynchronous transfer sends messages as quickly as
    // possible without waiting for confirmation.
    for (int i = 0; i < 10; i++)
    {
        message.clearData();
        message.appendData(Encoding.UTF8.GetBytes("Message " + i));
        session.messageTransfer("amq.topic", routing_key, message);
    }
}

```

```
}  
}
```

The `noMoreMessages` function signals the end of messages using the control routing key, which is reserved for control messages.

```
private static void noMoreMessages(ClientSession session)  
{  
    IMessage message = new Message();  
    // And send a synchrononous final message to indicate termination.  
    message.clearData();  
    message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));  
    session.messageTransfer("amq.topic", "control", message);  
    session.sync();  
}
```

In the main body of the program, messages are published using four different routing keys, and then the end of messages is indicated by a message sent to a separate routing key.

```
publishMessages(session, "usa.news");  
publishMessages(session, "usa.weather");  
publishMessages(session, "europe.news");  
publishMessages(session, "europe.weather");  
  
noMoreMessages(session);
```

### 7.3.5.2. Reading Messages from the Queue

The second program in the publish/subscribe example, **Listener.cs**, creates a local private queue, with a unique name, for each of the four binding keys it specifies: `usa.#`, `europe.#`, `#.news`, and `#.weather`, then creates a listener for those queues.

```
Console.WriteLine("Listening for messages ...");  
// Create a listener  
prepareQueue("usa", "usa.#", session);  
prepareQueue("europe", "europe.#", session);  
prepareQueue("news", "#.news", session);  
prepareQueue("weather", "#.weather", session);
```

The `prepareQueue()` method creates a queue using a queue name and a routing key supplied as arguments it then attaches a listener with the session for the created queue and subscribe for this receiving messages from the queue:

```
// Create a unique queue name for this consumer by concatenating  
// the queue name parameter with the Session ID.  
Console.WriteLine("Declaring queue: " + queue);
```

```

session.queueDeclare(queue, Option.EXCLUSIVE, Option.AUTO_DELETE);

// Route messages to the new queue if they match the routing key.
// Also route any messages to with the "control" routing key to
// this queue so we know when it's time to stop. A publisher sends
// a message with the content "That's all, Folks!", using the
// "control" routing key, when it is finished.

session.exchangeBind(queue, "amq.topic", routing_key);
session.exchangeBind(queue, "amq.topic", "control");

// subscribe the listener to the queue
IMessageListener listener = new MessageListener(session);
session.attachMessageListener(listener, queue);
session.messageSubscribe(queue);

```

### 7.3.5.3. Writing Request/Response Applications

In the request/response examples, we write a server that accepts strings from clients and converts them to upper case, sending the result back to the requesting client. This example consists of two programs.

- **Client.cs** is a client application that sends messages to the server.
- **Server.cs** is a service that accepts messages, converts their content to upper case, and sends the result to the **amq.direct** exchange, using the request's **reply-to** property as the routing key for the response.

### 7.3.5.4. Running the Request/Response Examples

1. Make sure your **PATH** contains the directory **<home>/qpidd/lib**
2. Make sure that a qpidd broker is running:

```
$ ps -eaf | grep qpidd
```

If a broker is running, you should see the qpidd process in the output of the above command.

3. Run the server.

- cygwin

```

$ cd <home>/qpidd/examples/direct
$ ./example-request-response-Server.exe [hostname] [portnumber]

```

- mono

```
$ cd <home>/qpidd/examples/direct
```

```
$ mono ./example-request-response-Server.exe [hostname] [portnumber]
```

You will see output similar to this:

```
Waiting for requests
```

4. In a separate window, start a client:

- cygwin

```
$ cd <home>/qpid/examples/direct
$ ./example-request-response-Client.exe [hostname] [portnumber]
```

- mono

```
$ cd <home>/qpid/examples/direct
$ mono ./example-request-response-Client.exe [hostname] [portnumber]
```

5. You will see output similar to this:

```
Activating response queue listener for: clientSystem.Byte[]
Waiting for all responses to arrive ...
Response: TWAS BRILLIG, AND THE SLITHY TOVES
Response: DID GIRE AND GYMBLE IN THE WABE.
Response: ALL MIMSY WERE THE BOROGROVES,
Response: AND THE MOME RATHS OUTGRABE.
Shutting down listener for clientSystem.Byte[]
Response: THAT'S ALL, FOLKS!
```

6. Go back to the server window, the output should be similar to this:

```
Waiting for requests
Request: Twas brillig, and the slithy toves
Request: Did gire and gymbble in the wabe.
Request: All mimsy were the borogroves,
Request: And the mome raths outgrabe.
Request: That's all, folks!
```

Now we will examine the code for each of these programs. In each section, we will discuss only the code that must be added to the skeleton shown in [Section 5.1, "Creating and Closing Sessions"](#).

### 7.3.5.5. The Client Application

The first program in the request-response example, `Client.cs`, sets up a private response queue to receive responses from the server, then sends messages the server, listening to the response queue for the server's responses.

First, let's create a response queue and bind it to the **amq.direct** exchange. We will use the session id both as the name of the queue and as the routing key.

```
string response_queue = "client" + session.getName();

session.queueDeclare(response_queue);
session.exchangeBind(response_queue, "amq.direct", response_queue);
```

Now let's create a listener for the response queue and listen for messages.

```
Console.WriteLine("Activating response queue listener for: " +
    response_queue);
IMessageListener listener = new ClientMessageListener(session);
session.attachMessageListener(listener, response_queue);
session.messageSubscribe(response_queue);
```

When we send requests, we will always send it to the **request** queue on the **amq.direct** exchange. We will also place the routing key for our private queue in the reply-to field of the message. Let's set those properties for a message:

```
IMessage request = new Message();
request.DeliveryProperties.setRoutingKey("request");
request.MessageProperties.setReplyTo(new ReplyTo("amq.direct",
    response_queue));
```

Now let's send some requests...

```
string[] strs = {
    "Twas brillig, and the slithy toves",
    "Did gire and gymble in the wabe.",
    "All mimsy were the borogroves,",
    "And the mome raths outgrabe.",
    "That's all, folks!"
};
foreach (string s in strs)
{
    request.clearData();
    request.appendData(Encoding.UTF8.GetBytes(s));
    session.messageTransfer("amq.direct", request);
}
```

And wait for responses to arrive:

```
Console.WriteLine("Waiting for all responses to arrive ...");  
Monitor.Wait(session);
```

### 7.3.5.6. The Server Application

The second program in the request-response example, `Server.cs`, uses the `reply-to` property as the routing key for responses.

The main body of `Server.cs` creates an exclusive queue for requests, then waits for messages to arrive.

```
const string request_queue = "request";  
// Use the name of the request queue as the routing key  
session.queueDeclare(request_queue);  
session.exchangeBind(request_queue, "amq.direct", request_queue);  
  
lock (session)  
{  
    // Create a listener and subscribe it to the request_queue  
    IMessageListener listener = new MessageListener(session);  
    session.attachMessageListener(listener, request_queue);  
    session.messageSubscribe(request_queue);  
  
    // Receive messages until all messages are received  
    Console.WriteLine("Waiting for requests");  
    Monitor.Wait(session);  
}
```

The listener's `messageTransfer()` method converts the request's content to upper case, then sends a response to the broker, using the request's `reply-to` property as the routing key for the response.

```
BinaryReader reader = new BinaryReader(request.Body, Encoding.UTF8);  
byte[] body = new byte[request.Body.Length - request.Body.Position];  
reader.Read(body, 0, body.Length);  
ASCIIEncoding enc = new ASCIIEncoding();  
string message = enc.GetString(body);  
Console.WriteLine("Request: " + message);  
  
// Transform message content to upper case  
string responseBody = message.ToUpper();  
  
// Send it back to the user  
response.clearData();  
response.appendData(Encoding.UTF8.GetBytes(responseBody));  
_session.messageTransfer("amq.direct", routingKey, response);
```

---

# Appendix A. Revision History

Revision 1.1      Fri Sep 5 2008      Lana Brindley [lbrindle@redhat.com](mailto:lbrindle@redhat.com)  
Fixed build error

Revision 1.0      Thu Sep 4 2008      Lana Brindley [lbrindle@redhat.com](mailto:lbrindle@redhat.com)  
Updated document for new build system

