Red Hat Enterprise MRG 2.0

Programming in Apache Qpid

Cross-Platform AMQP Messaging in C++, Python, Java JMS and .NET

Jonathan Robie

Chuck Rolke

Alison Young

Red Hat Enterprise MRG 2.0 Programming in Apache Qpid Cross-Platform AMQP Messaging in C++, Python, Java JMS and .NET
Edition 1

Author	Jonathan Robie	
Author	Chuck Rolke	`crolke@redhat.com`
Author	Alison Young	`alyoung@redhat.com`

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

This book describes how to write programs for MRG Messaging using the Apache Qpid Messaging and Java JMS APIs.

Preface

1. Document Conventions

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

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

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

`Mono-spaced Bold`

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

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

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

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

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

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

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

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

Proportional Bold

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

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

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

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

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

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

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

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

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

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

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

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

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

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

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

import javax.naming.InitialContext;
```

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

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

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

1.3. Notes and Warnings

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

Note

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

Important

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

Warning

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

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

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

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

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

2.2. We Need Feedback!

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

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

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

Introduction

Apache Qpid is a reliable, asynchronous messaging system that supports the AMQP messaging protocol in several common programming languages. Qpid is supported on most common platforms.

- On the Java platform, Qpid uses the established *Java JMS API*[1].
- For Python , C++ and .NET, Qpid defines its own messaging API, the *Qpid Messaging API*, which is conceptually similar in each supported language.
- Support for this API in Ruby will be added soon (Ruby currently uses an API that is closely tied to the AMQP version).

Using the Qpid Messaging API

The Qpid Messaging API is quite simple, consisting of only a handful of core classes.

- A *message* consists of a standard set of fields (such as **`subject`**, **`reply-to`**), an application-defined set of properties, and message content (the main body of the message).
- A *connection* represents a network connection to a remote endpoint.
- A *session* provides a sequentially ordered context for sending and receiving messages. A session is obtained from a connection.
- A *sender* sends messages to a target using the **`sender.send`** method. A sender is obtained from a session for a given target address.
- A *receiver* receives messages from a source using the **`receiver.fetch`** method. A receiver is obtained from a session for a given source address.

The following sections show how to use these classes in a simple messaging program.

2.1. A Simple Messaging Program in C++

The following C++ program shows how to create a connection, create a session, send messages using a sender, and receive messages using a receiver.

Example 2.1. "Hello world!" in C++

```
#include <qpid/messaging/Connection.h>
#include <qpid/messaging/Message.h>
#include <qpid/messaging/Receiver.h>
#include <qpid/messaging/Sender.h>
#include <qpid/messaging/Session.h>

#include <iostream>

using namespace qpid::messaging;

int main(int argc, char** argv) {
    std::string broker = argc > 1 ? argv[1] : "localhost:5672";
    std::string address = argc > 2 ? argv[2] : "amq.topic";
    Connection connection(broker);
    try {
        connection.open();  «1»
        Session session = connection.createSession(); «2»

        Receiver receiver = session.createReceiver(address); «3»
        Sender sender = session.createSender(address); «4»

        sender.send(Message("Hello world!"));

        Message message = receiver.fetch(Duration::SECOND * 1); «5»
        std::cout << message.getContent() << std::endl;
        session.acknowledge(); «6»

        connection.close(); «7»
        return 0;
    } catch(const std::exception& error) {
        std::cerr << error.what() << std::endl;
        connection.close();
        return 1;
    }
      }
```

«1» Establishes the connection with the messaging broker.

«2» Creates a session object on which messages will be sent and received.

«3» Creates a receiver that receives messages from the given address.

«4» Creates a sender that sends to the given address.

«5» Receives the next message. The duration is optional. If omitted, it will wait indefinitely for the next message.

«6» Acknowledges receipt of all fetched messages on the session. This informs the broker that the messages were transferred and processed by the client successfully.

«7» Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

2.2. A Simple Messaging Program in Python

The following Python program shows how to create a connection, create a session, send messages using a sender, and receive messages using a receiver.

Example 2.2. "Hello world!" in Python

```
import sys
from qpid.messaging import *

broker =  "localhost:5672" if len(sys.argv)<2 else sys.argv[1]
address = "amq.topic" if len(sys.argv)<3 else sys.argv[2]

connection = Connection(broker)

try:
  connection.open()  «1»
  session = connection.session()   «2»

  sender = session.sender(address)  «3»
  receiver = session.receiver(address)  «4»

  sender.send(Message("Hello world!"));

  message = receiver.fetch(timeout=1)  «5»
  print message.content
  session.acknowledge() «6»

except MessagingError,m:
  print m
finally:
  connection.close()  «7»
```

«1» Establishes the connection with the messaging broker.

«2» Creates a session object on which messages will be sent and received.

«3» Creates a sender that sends to the given address.

«4» Creates a receiver that receives messages from the given address.

«5» Receives the next message. The duration is optional. If omitted, it will wait indefinitely for the next message.

«6» Acknowledges receipt of all fetched messages on the session. This informs the broker that the messages were transferred and processed by the client successfully.

«7» Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

2.3. A Simple Messaging Program in .NET C#

The following .NET C# [1] program shows how to create a connection, create a session, send messages using a sender, and receive messages using a receiver.

Example 2.3. "Hello world!" in .NET C#

```
using System;
using Org.Apache.Qpid.Messaging;  «1»

namespace Org.Apache.Qpid.Messaging {
    class Program {
        static void Main(string[] args) {
            String broker = args.Length > 0 ? args[0] : "localhost:5672";
            String address = args.Length > 1 ? args[1] : "amq.topic";

            Connection connection = null;
            try {
                connection = new Connection(broker);
                connection.Open();   «2»
                Session session = connection.CreateSession();   «3»

                Receiver receiver = session.CreateReceiver(address);   «4»
                Sender sender = session.CreateSender(address);   «5»

                sender.Send(new Message("Hello world!"));

                Message message = new Message();
                message = receiver.Fetch(DurationConstants.SECOND * 1);   «6»
                Console.WriteLine("{0}", message.GetContent());
                session.Acknowledge();   «7»

                connection.Close();   «8»
            } catch (Exception e) {
                Console.WriteLine("Exception {0}.", e);
                if (connection != null)
                    connection.Close();
            }
        }
    }
}
```

«1» Permits use of **`Org.Apache.Qpid.Messaging`** types and methods without explicit namespace qualification. Any .NET project must have a project reference to the assembly file **`Org.Apache.Qpid.Messaging.dll`** in order to obtain the definitions of the .NET Binding for Qpid Messaging namespace.

[1] The .NET binding for the Qpid C++ Messaging API applies to all .NET Framework managed code languages. C# was chosen for illustration purposes only.

«2» Establishes the connection with the messaging broker.

«3» Creates a session object on which messages will be sent and received.

«4» Creates a receiver that receives messages from the given address.

«5» Creates a sender that sends to the given address.

«6» Receives the next message. The duration is optional. If omitted, it will wait indefinitely for the next message.

«7» Acknowledges receipt of all fetched messages on the session. This informs the broker that the messages were transferred and processed by the client successfully.

«8» Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

2.4. Addresses

An *address* is the name of a message target or message source. [2] The methods that create senders and receivers require an address. The details of sending to a particular target or receiving from a particular source are then handled by the sender or receiver. A different target or source can be used by specifying a different address.

An address resolves to a *node*. The Qpid Messaging API recognizes two kinds of nodes, *queues* and *topics* [3]. A queue stores each message until it has been received and acknowledged, and only one receiver can receive a given message [4] A topic immediately delivers a message to all eligible receivers. If there are no eligible receivers, it discards the message. In the AMQP 0-10 implementation of the API, [5] queues map to AMQP queues, and topics map to AMQP exchanges. [6]

This tutorial contains many examples that use two programs that take an address as a command line parameter. **`Spout`** sends messages to the target address, **`drain`** receives messages from the source address. The source code is available in C++, Python, and .NET C#, and can be found in the examples directory for each language. These programs can use any address string as a source or a destination, and have many command line options to configure behavior. Use the **`-h`** option for documentation on these options. [7] The examples in this tutorial also use the **`qpid-config`** utility to configure AMQP 0-10 queues and exchanges on a Qpid broker.

Example 2.4. Address as Queue

Create a queue with **`qpid-config`**, send a message using **`spout`**, and read it using **`drain`**:

[2] In the example programs, the **`amq.topic`** is used as the default address if none is passed in. This is the name of a standard exchange that always exists on an AMQP 0-10 messaging broker.
[3] The terms *queue* and *topic* were chosen to align with their meaning in JMS. These two addressing patterns are sometimes referred as *point-to-point* and *publish-subscribe* instead. AMQP 0-10 has an exchange type called a *topic exchange*. When the term *topic* occurs alone, it refers to a Messaging API topic, not the topic exchange.
[4] There are exceptions to this rule; for instance, a receiver can use **`browse`** mode, which leaves messages on the queue for other receivers to read.
[5] The AMQP 0-10 implementation is the only one that currently exists.
[6] In AMQP 0-10, messages are sent to exchanges, and read from queues. The Messaging API also allows a sender to send messages to a queue; internally, Qpid implements this by sending the message to the default exchange, with the name of the queue as the routing key. The Messaging API also allows a receiver to receive messages from a topic; internally, Qpid implements this by setting up a private subscription queue for the receiver and binding the subscription queue to the exchange that corresponds to the topic.
[7] Currently, the Python, C++, and .NET C# implementations of **`drain`** and **`spout`** have slightly different options. This tutorial uses the C++ implementation. The options will be reconciled in the near future.

```
$ qpid-config add queue hello-world
$ ./spout hello-world
$ ./drain hello-world

Message(properties={spout-id:c877e622-d57b-4df2-bf3e-6014c68da0ea:0}, content='')
```

The queue stored the message sent by **spout** and delivered it to **drain** when requested.

Once the message has been delivered and acknowledged by **drain**, it is no longer available on the queue. If we run **drain** one more time, no messages will be retrieved.

```
$ ./drain hello-world
$
```

Example 2.5. Address as Topic

This example is similar to the previous example, but it uses a topic instead of a queue.

First, use **qpid-config** to remove the queue and create an exchange with the same name:

```
$ qpid-config del queue hello-world
$ qpid-config add exchange topic hello-world
```

Now run **drain** and **spout** the same way we did in the previous example:

```
$ ./spout hello-world
$ ./drain hello-world
$
```

Topics deliver messages immediately to any interested receiver, and do not store messages. Because there were no receivers at the time **spout** sent the message, it was simply discarded. When the **drain** was run, there were no messages to receive.

Now run **drain** first, using the *-t* option to specify a timeout in seconds. While **drain** is waiting for messages, run **spout** in another window.

First Window:

```
$ ./drain -t 30 hello-word
```

Second Window:

```
$ ./spout hello-word
```

Once **spout** has sent a message, return to the first window to see the output from **drain**:

```
Message(properties={spout-id:7da2d27d-93e6-4803-8a61-536d87b8d93f:0}, content='')
```

You can run **drain** in several separate windows; each creates a subscription for the exchange, and each receives all messages sent to the exchange.

2.4.1. Address Strings

So far, our examples have used address strings that contain only the name of a node. An *address string* can also contain a *subject* and *options*.

The syntax for an address string is:

```
address_string ::=  <address> [ / <subject> ] [ ; <options> ]
options ::=  { <key> : <value>, ... }
```

Addresses, subjects, and keys are strings. Values can be numbers, strings (with optional single or double quotes), maps, or lists. A complete BNF for address strings appears in *Section 2.4.4, "Address String Grammar"*.

So far, the address strings in this tutorial have used only simple names. The following sections show how to use subjects and options.

2.4.2. Message Subjects

Every message has a property called *subject*, which is analogous to the subject on an email message. If no subject is specified, the message's subject is null. For convenience, address strings also allow a subject. If a sender's address contains a subject, it is used as the default subject for the messages it sends. If a receiver's address contains a subject, it is used to select only messages that match the subject—the matching algorithm depends on the message source.

In AMQP 0-10, each exchange type has its own matching algorithm. This is discussed in *Section 2.17, "The AMQP 0-10 mapping"*.

Note

Currently, a receiver bound to a queue ignores subjects, receiving messages from the queue without filtering. Support for subject filtering on queues will be implemented soon.

Example 2.6. Using subjects

This example shows how subjects affect message flow.

First, use **`qpid-config`** to create a topic exchange.

```
$ qpid-config add exchange topic news-service
```

Now use drain to receive messages from *`news-service`* that match the subject *`sports`*:

First Window:

```
$ ./drain -t 30 news-service/sports
```

In a second window, send messages to *`news-service`* using two different subjects:

Second Window:

```
$ ./spout news-service/sports
$ ./spout news-service/news
```

Now look at the first window, the message with the subject *sports* has been received, but not the message with the subject *news*:

```
Message(properties={qpid.subject:sports, spout-id:9441674e-a157-4780-a78e-f7ccea998291:0},
 content='')
```

If you run **drain** in multiple windows using the same subject, all instances of **drain** receive the messages for that subject.

The AMQP exchange type used here, *amq.topic*, can also do more sophisticated matching. A sender's subject can contain multiple words separated by a "." delimiter. For instance, in a news application, the sender might use subjects like *usa.news*, *usa.weather*, *europe.news*, or *europe.weather*. The receiver's subject can include wildcard characters — "#" matches one or more words in the message's subject, "*" matches a single word. For instance, if the subject in the source address is **.news*, it matches messages with the subject *europe.news* or *usa.news*; if it is *europe.#*, it matches messages with subjects like *europe.news* or *europe.pseudo.news*.

Example 2.7. Subjects with multi-word keys

This example uses **drain** and **spout** to demonstrate the use of subjects with two-word keys.

Use **drain** with the subject **.news* to listen for messages in which the second word of the key is *news*.

First Window:

```
$ ./drain -t 30 news-service/*.news
```

Now send messages using several different two-word keys:

Second Window:

```
$ ./spout news-service/usa.news
$ ./spout news-service/usa.sports
$ ./spout news-service/europe.sports
$ ./spout news-service/europe.news
```

In the first window, the messages with *news* in the second word of the key have been received:

```
Message(properties={qpid.subject:usa.news, spout-id:73fc8058-5af6-407c-9166-
b49a9076097a:0}, content='')
Message(properties={qpid.subject:europe.news, spout-id:f72815aa-7be4-4944-99fd-
c64c9747a876:0}, content='')
```

Next, use **drain** with the subject *#.news* to match any sequence of words that ends with *news*.

First Window:

```
$ ./drain -t 30 news-service/#.news
```

In the second window, send messages using a variety of different multi-word keys:

Second Window:

```
$ ./spout news-service/news
$ ./spout news-service/sports
$ ./spout news-service/usa.news
$ ./spout news-service/usa.sports
$ ./spout news-service/usa.faux.news
$ ./spout news-service/usa.faux.sports
```

In the first window, messages with *news* in the last word of the key have been received:

```
Message(properties={qpid.subject:news, spout-id:cbd42b0f-c87b-4088-8206-26d7627c9640:0},
 content='')
Message(properties={qpid.subject:usa.news, spout-id:234a78d7-
daeb-4826-90e1-1c6540781eac:0}, content='')
Message(properties={qpid.subject:usa.faux.news, spout-id:6029430a-cfcb-4700-8e9b-
cbe4a81fca5f:0}, content='')
```

2.4.3. Address String Options

The options in an address string can contain additional information for the senders or receivers created for it, including:

- Policies for assertions about the node to which an address refers.

  For instance, in the address string *my-queue; {assert: always, node:{ type: queue }}*, the node named *my-queue* must be a queue; if not, the address does not resolve to a node, and an exception is raised.

- Policies for automatically creating or deleting the node to which an address refers.

  For instance, in the address string *xoxox ; {create: always}*, the queue *xoxox* is created, if it does not exist, before the address is resolved.

- Extension points that can be used for sender/receiver configuration.

  For instance, if the address for a receiver is *my-queue; {mode: browse}*, the receiver works in *browse* mode, leaving messages on the queue so other receivers can receive them.

- Extension points that provide more direct control over the underlying protocol.

  For instance, the *x-bindings* property allows greater control over the AMQP 0-10 binding process when an address is resolved.

The following examples show how these different kinds of address string options affect the behavior of senders and receivers.

2.4.3.1. *assert*

In this section, we use the *assert* option to ensure that the address resolves to a node of the required type.

Example 2.8. Assertions on Nodes

Use **qpid-config** to create a queue and a topic.

```
$ qpid-config add queue my-queue
$ qpid-config add exchange topic my-topic
```

Now use the address specified to **`drain`** to assert that it is of a particular type:

```
$ ./drain 'my-queue; {assert: always, node:{ type: queue }}'
$ ./drain 'my-queue; {assert: always, node:{ type: topic }}'
2010-04-20 17:30:46 warning Exception received from broker: not-found: not-found: Exchange
 not found: my-queue (../../src/qpid/broker/ExchangeRegistry.cpp:92) [caused by 2 \x07:
\x01]
Exchange my-queue does not exist
```

The first attempt passed without error as *`my-queue`* is a queue. The second attempt however failed; *`my-queue`* is not a topic.

The same thing can be done for *`my-topic`*:

```
$ ./drain 'my-topic; {assert: always, node:{ type: topic }}'
$ ./drain 'my-topic; {assert: always, node:{ type: queue }}'
2010-04-20 17:31:01 warning Exception received from broker: not-found: not-found: Queue
 not found: my-topic (../../src/qpid/broker/SessionAdapter.cpp:754) [caused by 1 \x08:
\x01]
Queue my-topic does not exist
```

2.4.3.2. *`create`*

In previous examples, the queue was created before listening for messages on it. Using *`create: always`*, the queue is automatically created if it does not exist.

Example 2.9. Creating a Queue Automatically

First Window:

```
$ ./drain -t 30 "xoxox ; {create: always}"
```

Now send messages to this queue:

Second Window:

```
$ ./spout "xoxox ; {create: always}"
```

Returning to the first window, the **`drain`** has received this message:

```
Message(properties={spout-id:1a1a3842-1a8b-4f88-8940-b4096e615a7d:0}, content='')
```

The details of the node thus created can be controlled by further options within the node. See *Table 2.2, "Node Properties"* for details.

2.4.3.3. *`browse`*

Some options specify message transfer semantics. For example, they might state whether messages should be consumed or read in browsing mode, or specify reliability characteristics. The following example uses the *`browse`* option to receive messages without removing them from a queue.

Example 2.10. Browsing a Queue

Use the browse mode to receive messages without removing them from the queue. First, send three messages to the queue:

```
$ ./spout my-queue --content one
$ ./spout my-queue --content two
$ ./spout my-queue --content three
```

Now use drain to get those messages, using the browse option:

```
$ ./drain 'my-queue; {mode: browse}'
Message(properties={spout-id:fbb93f30-0e82-4b6d-8c1d-be60eb132530:0}, content='one')
Message(properties={spout-id:ab9e7c31-19b0-4455-8976-34abe83edc5f:0}, content='two')
Message(properties={spout-id:ea75d64d-ea37-47f9-96a9-d38e01c97925:0}, content='three')
```

You can confirm the messages are still on the queue by repeating the drain:

```
$ ./drain 'my-queue; {mode: browse}'
Message(properties={spout-id:fbb93f30-0e82-4b6d-8c1d-be60eb132530:0}, content='one')
Message(properties={spout-id:ab9e7c31-19b0-4455-8976-34abe83edc5f:0}, content='two')
Message(properties={spout-id:ea75d64d-ea37-47f9-96a9-d38e01c97925:0}, content='three')
```

2.4.3.4. *`x-bindings`*

Greater control over the AMQP 0-10 binding process can be achieved by including an *`x-bindings`* option in an address string. For example, the XML Exchange is an AMQP 0-10 custom exchange provided by the Apache Qpid C++ broker. It allows messages to be filtered using XQuery; queries can address either message properties or XML content in the body of the message. These queries can be specified in addresses using x-bindings

An instance of the XML Exchange must be added before it can be used:

```
$ qpid-config add exchange xml xml
```

When using the XML Exchange, a receiver provides an XQuery as an x-binding argument. If the query contains a context item (a path starting with “.”), then it is applied to the content of the message, which must be well-formed XML. For instance, *`./weather`* is a valid XQuery, which matches any message in which the root element is named *`weather`*. Here is an address string that contains this query:

```
xml; {
 link: {
  x-bindings: [{exchange:xml, key:weather, arguments:{xquery:"./weather"} }]
 }
}
```

When using longer queries with **`drain`**, it is often useful to place the query in a file, and use **`cat`** in the command line. We do this in the following example.

Example 2.11. Using the XML Exchange

This example uses an x-binding that contains queries, which filter based on the content of XML messages. Here is an XQuery for this example:

```
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
```

This query can be specified in an x-binding to listen to messages that meet the criteria specified by the query:

First Window:

```
$ ./drain -f "xml; {link:{x-bindings:[{key:'weather',
arguments:{xquery:\"$(cat rdu.xquery )\"}}]}}"
```

In another window, create an XML message that meets the criteria in the query, and place it in the file **`rdu.xml`**:

```
<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 use **`spout`** to send this message to the XML exchange:

Second Window:

```
spout --content "$(cat rdu.xml)" xml/weather
```

Returning to the first window, we see that the message has been received:

```
$ ./drain -f "xml; {link:{x-bindings:[{exchange:'xml', key:'weather', arguments:{xquery:
\"$(cat rdu.xquery )\"}}]}}"
Message(properties={qpid.subject:weather, spout-id:31c431de-593f-4bec-
a3dd-29717bd945d3:0},
content='<weather>
  <station>Raleigh-Durham International Airport (KRDU)</station>
  <wind_speed_mph>16</wind_speed_mph>
  <temperature_f>40</temperature_f>
  <dewpoint>35</dewpoint>
</weather>')
```

2.4.3.5. Address String Options - Reference

Table 2.1. Address String Options

Option	Value	Semantics
`assert`	*`always`*, *`never`*, *`sender`*, or *`receiver`*	Asserts that the properties specified in the node option match whatever the address

Option	Value	Semantics
		resolves to. If they do not, resolution fails and an exception is raised.
`create`	*`always`*, *`never`*, *`sender`*, or *`receiver`*	Creates the node to which an address refers if it does not exist. No error is raised if the node does exist. The details of the node may be specified in the node option.
`delete`	*`always`*, *`never`*, *`sender`*, or *`receiver`*	Delete the node when the sender or receiver is closed.
`node`	A nested map containing the entries shown in *Table 2.2, "Node Properties"*.	Specifies properties of the node to which the address refers. These are used in conjunction with the assert or create options.
`link`	A nested map containing the entries shown in *Table 2.3, "Link Properties"*.	Used to control the establishment of a conceptual link from the client application to or from the target/source address.
`mode`	*`browse`* or *`consume`*	This option is only of relevance for source addresses that resolve to a queue. If browse is specified the messages delivered to the receiver are left on the queue rather than being removed. If consume is specified the normal behavior applies; messages are removed from the queue once the client acknowledges their receipt.

Table 2.2. Node Properties

Property	Value	Semantics
`type`	*`topic`* or *`queue`*	Indicates the type of the node.
`durable`	*`True`* or *`False`*	Indicates whether the node survives a loss of volatile storage e.g. if the broker is restarted.
`x-declare`	A nested map whose values correspond to the valid fields on an AMQP 0-10 queue-declare or exchange-declare command.	These values are used to fine tune the creation or assertion process. Note however that they are protocol specific.
`x-bindings`	A nested list in which each binding is represented by a map. The entries of the map	In conjunction with the create option, each of these bindings is established as the address

Property	Value	Semantics
	for a binding contain the fields that describe an AMQP 0-10 binding. The format for x-bindings is as follows: `[ { exchange: <exchange>, queue: <queue>, key: <key>, arguments: { <key_1>: <value_1>, ..., <key_n>: <value_n> } }, ... ]`	is resolved. In conjunction with the assert option, the existence of each of these bindings is verified during resolution. These are protocol specific.

Table 2.3. Link Properties

Option	Value	Semantics
`reliability`	*`unreliable`*, *`at-least-once`*, *`at-most-once`*, or *`exactly-once`*	Reliability indicates the level of reliability that the sender or receiver. *`unreliable`* and *`at-most-once`* are currently treated as synonyms, and allow messages to be lost if a broker crashes or the connection to a broker is lost. *`at-least-once`* guarantees that a message is not lost, but duplicates may be received. *`exactly-once`* guarantees that a message is not lost, and is delivered precisely once. Currently only *`unreliable`* and *`at-least-once`* are supported. [1]
`durable`	*`True`* or *`False`*	Indicates whether the link survives a loss of volatile storage, for example if the broker is restarted.
`x-declare`	A nested map whose values correspond to the valid fields of an AMQP 0-10 queue-declare command.	These values can be used to customize the subscription queue in the case of receiving from an exchange. Note however that they are protocol specific.
`x-subscribe`	A nested map whose values correspond to the valid fields of an AMQP 0-10 message-subscribe command.	These values can be used to customize the subscription.
`x-bindings`	A nested list each of whose entries is a map that may	These bindings are established during resolution independent

Option	Value	Semantics
	contain fields (queue, exchange, key and arguments) describing an AMQP 0-10 binding.	of the create option. They are considered logically part of the linking process rather than of node creation.

[1] If `at-most-once` is requested, unreliable will be used and for durable messages on durable queues there is the possibility that messages will be redelivered; if exactly-once is requested, at-most-once will be used and the application needs to be able to deal with duplicates.

2.4.4. Address String Grammar

This section provides a formal grammar for address strings.

Tokens

The following regular expressions define the tokens used to parse address strings:

```
LBRACE: \\{
RBRACE: \\}
LBRACK: \\[
RBRACK: \\]
COLON:  :
SEMI:   ;
SLASH:  /
COMMA:  ,
NUMBER: [+-]?[0-9]*\\.?[0-9]+
ID:     [a-zA-Z_](?:[a-zA-Z0-9_-]*[a-zA-Z0-9_])?
STRING: "(?:[^\\\\"]|\\\\.)*"|\'(?:[^\\\\\']|\\\\.)*\'
ESC:    \\\\[^ux]|\\\\x[0-9a-fA-F][0-9a-fA-F]|\\\\u[0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-
F]
SYM:    [.#*%@$^!+-]
WSPACE: [ \\n\\r\\t]+
```

Grammar

The formal grammar for addresses is as follows:

```
address := name [ SLASH subject ] [ ";" options ]
   name := ( part | quoted )+
subject := ( part | quoted | SLASH )*
 quoted := STRING / ESC
   part := LBRACE / RBRACE / COLON / COMMA / NUMBER / ID / SYM
options := map
    map := "{" ( keyval ( "," keyval )* )? "}"
 keyval "= ID ":" value
  value := NUMBER / STRING / ID / map / list
   list := "[" ( value ( "," value )* )? "]"
```

Address String Options

The address string options map supports the following parameters:

```
<name> [ / <subject> ] ; {
  create: always | sender | receiver | never,
  delete: always | sender | receiver | never,
  assert: always | sender | receiver | never,
  mode: browse | consume,
  node: {
    type: queue | topic,
    durable: True | False,
    x-declare: { ... <declare-overrides> ... },
    x-bindings: [<binding_1>, ... <binding_n>]
  },
  link: {
    name: <link-name>,
    durable: True | False,
    reliability: unreliable | at-most-once | at-least-once | exactly-once,
    x-declare: { ... <declare-overrides> ... },
    x-bindings: [<binding_1>, ... <binding_n>],
    x-subscribe: { ... <subscribe-overrides> ... }
  }
}
```

Create, Delete, and Assert Policies

The *`create`*, *`delete`*, and *`assert`* policies specify who should perform the associated action:

- **`always`**: the action is performed by any messaging client
- **`sender`**: the action is only performed by a sender
- **`receiver`**: the action is only performed by a receiver
- **`never`**: the action is never performed (this is the default)

Node-Type

The node-type is one of:

- **`topic`**: in the AMQP 0-10 mapping, a topic node defaults to the topic exchange, x-declare may be used to specify other exchange types
- **`queue`**: this is the default node-type

2.5. Sender Capacity and Replay

The *send* method of a sender has an optional second parameter that controls whether the *send* call is synchronous or not. A synchronous *send* call will block until the broker has confirmed receipt of the message. An asynchronous *send* call will return before the broker confirms receipt of the message, allowing for example further *send* calls to be made without waiting for a roundtrip to the broker for each message. This is desirable where increased throughput is important.

The sender maintains a list of sent messages whose receipt has yet to be confirmed by the broker. The maximum number of such messages that it will hold is defined by the capacity of the sender, which can be set by the application. If an application tries to send with a sender whose capacity is already fully used, the *send* call will block waiting for capacity regardless of the value of the sync flag.

The sender can be queried for the available space (the unused capacity), and for the current count of unsettled messages (those held in the replay list pending confirmation by the server). When the unsettled count is zero, all messages on that sender have been successfully sent.

If the connection fails and is transparently reconnected (see *Section 2.10, “Connection Options”* for details on how to control this feature), the unsettled messages for each sender over that connection will be re-transmitted. This provides a transparent level of reliability. This feature can be controlled through the link's reliability as defined in the address (see *Table 2.3, “Link Properties”*). At present only at-least-once guarantees are offered.

2.6. Receiver Capacity (Prefetch)

By default, a receiver requests the next message from the server in response to each fetch call, resulting in messages being sent to the receiver one at a time. As in the case of sending, it is often desirable to avoid this roundtrip for each message. This can be achieved by allowing the receiver to *prefetch* messages in anticipation of fetch calls being made. The receiver needs to be able to store these prefetched messages, the number it can hold is controlled by the receivers capacity.

2.7. Acknowledging Received Messages

Applications that receive messages should acknowledge their receipt by calling the session's *`acknowledge`* method. As in the case of sending messages, acknowledged transfer of messages to receivers provides at-least-once reliability, which means that the loss of the connection or a client crash does not result in lost messages; durable messages are not lost even if the broker is restarted. Some cases may not require this however and the reliability can be controlled through a link property in the address options (see *Table 2.3, “Link Properties”*).

The *`acknowledge`* call acknowledges all messages received on the session (i.e. all message that have been returned from a fetch call on a receiver created on that session).

The *`acknowledge`* call also supports an optional parameter controlling whether the call is synchronous or not. A synchronous *`acknowledge`* will block until the server has confirmed that it has received the acknowledgment. In the asynchronous case, when the call returns there is not yet any guarantee that the server has received and processed the acknowledgment. The session may be queried for the number of unsettled acknowledgments; when that count is zero all acknowledgments made for received messages have been successful.

2.8. Receiving Messages from Multiple Sources

A receiver can only read from one source, but many programs need to be able to read messages from many sources. In the Qpid Messaging API, a program can ask a session for the “next receiver”; that is, the receiver that is responsible for the next available message. The following examples show how this is done in C++, Python, and .NET C#.

Note that to use this pattern prefetching must be enabled for each receiver of interest so that the broker will send messages before a *`fetch`* call is made. See *Section 2.6, “Receiver Capacity (Prefetch)”* for more on this.

Example 2.12. Receiving Messages from Multiple Sources

C++:

```
Receiver receiver1 = session.createReceiver(address1);
receiver1.setCapacity(10);
Receiver receiver2 = session.createReceiver(address2);
receiver2.setCapacity(10);

Message message =  session.nextReceiver().fetch();
std::cout << message.getContent() << std::endl;
session.acknowledge(); // acknowledge message receipt
```

Python:

```
receiver1 = session.receiver(address1)
receiver1.capacity = 10
receiver2 = session.receiver(address)
```

```
receiver2.capacity = 10
message = session.next_receiver().fetch()
print message.content
session.acknowledge()
```

.NET C#:

```
Receiver receiver1 = session.CreateReceiver(address1);
receiver1.SetCapacity(10);
Receiver receiver2 = session.CreateReceiver(address2);
receiver2.SetCapacity(10);

Message message = new Message();
message =  session.NextReceiver().Fetch();
Console.WriteLine("{0}", message.GetContent());
session.Acknowledge();
```

2.9. Transactions

Sometimes it is useful to be able to group messages transfers - sent and received - on a session into atomic grouping. This can be done be creating the session as *transactional*. On a transactional session sent messages only become available at the target address on commit. Likewise any received and acknowledged messages are only discarded at their source on commit [8] .

Example 2.13. Transactions

C++:

```
Connection connection(broker);
Session session =  connection.createTransactionalSession();
...
if (smellsOk())
   session.commit();
else
   session.rollback();
```

.NET C#:

```
Connection connection = new Connection(broker);
Session session =  connection.createTransactionalSession();
...
if (smellsOk())
   session.Commit();
else
   session.Rollback();
```

2.10. Connection Options

Aspects of the connections behavior can be controlled through specifying connection options. For example, connections can be configured to automatically reconnect if the connection to a broker is lost.

[8] Note that this currently is only true for messages received using a reliable mode, such as at-least-once. Messages sent by a broker to an unreliable receiver will be discarded immediately regardless of transactionality.

Example 2.14. Specifying Connection Options in C++ and Python

In C++, these options can be set using **`Connection::setOption()`** or by passing in a set of options to the constructor. The options can be passed in as a map or in string form:

```
Connection connection("localhost:5672", "{reconnect: true}");
try {
    connection.open();
    !!! SNIP !!!
```

or

```
Connection connection("localhost:5672");
connection.setOption("reconnect", true);
try {
    connection.open();
    !!! SNIP !!!
```

In Python, these options can be set as attributes of the connection or using named arguments in the **`Connection`** constructor:

```
connection = Connection("localhost:5672", reconnect=True)
try:
  connection.open()
  !!! SNIP !!!
```

or

```
connection = Connection("localhost:5672")
connection.reconnect = True
try:
  connection.Open()
  !!! SNIP !!!
```

In .NET, these options can be set using **`Connection.SetOption()`** or by passing in a set of options to the constructor. The options can be passed in as a map or in string form:

```
Connection connection= new Connection("localhost:5672", "{reconnect: true}");
try {
    connection.Open();
    !!! SNIP !!!
```

or

```
Connection connection = new Connection("localhost:5672");
connection.SetOption("reconnect", true);
try {
    connection.Open();
    !!! SNIP !!!
```

See the reference documentation for details in each language.

The following table lists the supported connection options.

Table 2.4. Connection Options

Option name	Value type	Semantics
`username`	string	The username to use when authenticating to the broker.
`password`	string	The password to use when authenticating to the broker.
`sasl_mechanisms`	string	The specific SASL mechanisms to use when authenticating to the broker as a space separated list.
`reconnect`	boolean	Transparently reconnect if the connection is lost.
`reconnect_timeount`	integer	Total number of seconds to continue reconnection attempts before giving up and raising an exception.
`reconnect_limit`	integer	Maximum number of reconnection attempts before giving up and raising an exception.
`reconnect_interval_min`	integer representing time in seconds	Minimum number of seconds between reconnection attempts. The first reconnection attempt is made immediately; if that fails, the first reconnection delay is set to the value of **`reconnect_interval_min`**; if that attempt fails, the reconnect interval increases exponentially until a reconnection attempt succeeds or **`reconnect_interval_max`** is reached.
`reconnect_interval_max`	integer representing time in seconds	Maximum reconnect interval.
`reconnect_interval`	integer representing time in seconds	Sets both **`reconnection_interval_min`** and **`reconnection_interval_max`** to the same value.
`heartbeat`	integer representing time in seconds	Requests that heartbeats be sent every N seconds. If two successive heartbeats are missed the connection is considered to be lost.
`protocol`	string	Sets the underlying protocol used. The default option is *`tcp`*.

Option name	Value type	Semantics
		To enable ssl, set to *ssl*. The C++ client additionally supports *rdma*.
tcp_nodelay	boolean	Set *tcp_no_delay*, i.e. disable Nagle algorithm.

2.11. Maps and Lists in Message Content

Many messaging applications need to exchange data across languages and platforms, using the native data types of each programming language.

The Qpid Messaging API supports **map** and **list** in message content. [9] [10] Specific language support for **map** and **list** objects are shown in the following table.

Table 2.5. Map and List Representation in Supported Languages

Language	map	list
Python	**dict**	**list**
C++	**Variant::Map**	**Variant::List**
Java	**MapMessage**	
.NET	**Dictionary<string, object>**	**Collection<object>**

In all languages, messages are encoded using AMQP's portable data types.

Note

Because of the differences in type systems among languages, the simplest way to provide portable messages is to rely on maps, lists, strings, 64-bit signed integers, and doubles for messages that need to be exchanged across languages and platforms.

2.11.1. Qpid Maps and Lists in Python

In Python, Qpid supports the **dict** and **list** types directly in message content. The following code shows how to send these structures in a message:

Example 2.15. Sending Qpid Maps and Lists in Python

```
from qpid.messaging import *
# !!! SNIP !!!

content = {'Id' : 987654321, 'name' : 'Widget', 'percent' : 0.99}
content['colours'] = ['red', 'green', 'white']
content['dimensions'] = {'length' : 10.2, 'width' : 5.1,'depth' : 2.0};
content['parts'] = [ [1,2,5], [8,2,5] ]
content['specs'] = {'colors' : content['colours'],
                    'dimensions' : content['dimensions'],
```

[9] Unlike JMS, there is not a specific message type for map messages.
[10] Note that the Qpid JMS client supports MapMessages whose values can be nested maps or lists. This is not standard JMS behavior.

```
                    'parts' : content['parts'] }
message = Message(content=content)
sender.send(message)
```

The following table shows the data types that can be sent in a Python map message, and the corresponding data types that will be received by clients in Java or C++.

Table 2.6. Python Data Types in Maps

Python Data Type	--> C++	--> Java
`bool`	`bool`	`boolean`
`int`	`int64`	`long`
`long`	`int64`	`long`
`float`	`double`	`double`
`unicode`	`string`	`java.lang.String`
`uuid`	`qpid::types::Uuid`	`java.util.UUID`
`dict`	`Variant::Map`	`java.util.Map`
`list`	`Variant::List`	`java.util.List`

2.11.2. Qpid Maps and Lists in C++

In C++, Qpid defines the the **`Variant::Map`** and **`Variant::List`** types, which can be encoded into message content. The following code shows how to send these structures in a message:

Example 2.16. Sending Qpid Maps and Lists in C++

```

using namespace qpid::types;

// !!! SNIP !!!

Message message;
Variant::Map content;
content["id"] = 987654321;
content["name"] = "Widget";
content["percent"] = 0.99;
Variant::List colours;
colours.push_back(Variant("red"));
colours.push_back(Variant("green"));
colours.push_back(Variant("white"));
content["colours"] = colours;

Variant::Map dimensions;
dimensions["length"] = 10.2;
dimensions["width"] = 5.1;
dimensions["depth"] = 2.0;
content["dimensions"]= dimensions;

Variant::List part1;
part1.push_back(Variant(1));
part1.push_back(Variant(2));
part1.push_back(Variant(5));

Variant::List part2;
part2.push_back(Variant(8));
part2.push_back(Variant(2));
part2.push_back(Variant(5));
```

```
Variant::List parts;
parts.push_back(part1);
parts.push_back(part2);
content["parts"]= parts;

Variant::Map specs;
specs["colours"] = colours;
specs["dimensions"] = dimensions;
specs["parts"] = parts;
content["specs"] = specs;

encode(content, message);
sender.send(message, true);
```

The following table shows the data types that can be sent in a C++ map message, and the corresponding data types that will be received by clients in Java and Python.

Table 2.7. C++ Data Types in Maps

C++ Data Type	--> Python	--> Java
bool	bool	boolean
uint16	int | long	short
uint32	int | long	int
uint64	int | long	long
int16	int | long	short
int32	int | long	int
int64	int | long	long
float	float	float
double	float	double
string	unicode	java.lang.String
qpid::types::Uuid	uuid	java.util.UUID
Variant::Map	dict	java.util.Map
Variant::List	list	java.util.List

2.11.3. Qpid Maps and Lists in .NET C#

The .NET binding for the Qpid Messaging API binds .NET managed data types to C++ **`Variant`** data types. The following code shows how to send **`Variant::Map`** and **`Variant::List`** structures in a message:

Example 2.17. Sending Qpid Maps and Lists in .NET C

```

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using Org.Apache.Qpid.Messaging;

namespace Org.Apache.Qpid.Messaging.examples
{
    class MapSender
    {
        // csharp.map.sender example
        //
        // Send an amqp/map message
        // The map message contains simple types, a nested amqp/map,
        // an ampq/list, and specific instances of each supported type.
        //
        static int Main(string[] args)
        {
            string url = "amqp:tcp:localhost:5672";
            string address = "message_queue; {create: always}";
            string connectionOptions = "";

            if (args.Length > 0)
                url = args[0];
            if (args.Length > 1)
                address = args[1];
            if (args.Length > 2)
                connectionOptions = args[2];

            //
            // Create and open an AMQP connection to the broker URL
            //
            Connection connection = new Connection(url, connectionOptions);
            connection.Open();

            //
            // Create a session and a sender
            //
            Session session = connection.CreateSession();
            Sender sender = session.CreateSender(address);

            //
            // Create structured content for the message.  This example builds a
            // map of items including a nested map and a list of values.
            //
            Dictionary<string, object> content = new Dictionary<string, object>();
            Dictionary<string, object> subMap = new Dictionary<string, object>();
            Collection<object> colors = new Collection<object>();

            // add simple types
            content["id"] = 987654321;
            content["name"] = "Widget";
            content["percent"] = 0.99;

            // add nested amqp/map
            subMap["name"] = "Smith";
            subMap["number"] = 354;
            content["nestedMap"] = subMap;

            // add an amqp/list
            colors.Add("red");
            colors.Add("green");
            colors.Add("white");
            // list contains null value
            colors.Add(null);
            content["colorsList"] = colors;

            // add one of each supported amqp data type
            bool mybool = true;
            content["mybool"] = mybool;

            byte mybyte = 4;
            content["mybyte"] = mybyte;

            UInt16 myUInt16 = 5 ;
            content["myUInt16"] = myUInt16;

            UInt32 myUInt32 = 6;
            content["myUInt32"] = myUInt32;

            UInt64 myUInt64 = 7;
            content["myUInt64"] = myUInt64;

            char mychar = 'h';
            content["mychar"] = mychar;

            Int16 myInt16 = 9;
            content["myInt16"] = myInt16;

            Int32 myInt32 = 10;
            content["myInt32"] = myInt32;

            Int64 myInt64 = 11;
            content["myInt64"] = myInt64;

            Single mySingle = (Single)12.12;
            content["mySingle"] = mySingle;

            Double myDouble = 13.13;
            content["myDouble"] = myDouble;

            Guid myGuid = new Guid("000102030405060708090a0b0c0d0e0f");
            content["myGuid"] = myGuid;

            content["myNull"] = null;

            //
            // Construct a message with the map content and send it synchronously
            // via the sender.
            //
            Message message = new Message(content);
            sender.Send(message, true);

            //
            // Wait until broker receives all messages.
            //
            session.Sync();

            //
            // Close the connection.
            //
            connection.Close();
            return 0;
        }
    }
}
```

The following table shows the mapping between data types in .NET and C++..

Table 2.8. Data Type Mapping between C++ and .NET binding

C++ Data Type	.NET binding
`void`	`nullptr`
`bool`	`bool`
`uint8`	`byte`
`uint16`	`UInt16`
`uint32`	`UInt32`
`uint64`	`UInt64`
`int16`	`char`

C++ Data Type	.NET binding
`int16`	`Int16`
`int32`	`Int32`
`int64`	`Int64`
`float`	`Single`
`double`	`Double`
`string`	`string`
`qpid::types::Uuid`	`Guid`
`Variant::Map`	`Dictionary< string, object >`
`Variant::List`	`Collection< object >`

Note

.NET **`string`** objects are translated to and from C++ strings using UTF-8 encoding only.

2.12. The Request/Response Pattern

Request/Response applications use the *`reply-to`* property, described in *Table 2.10, "Mapping to AMQP 0-10 Message Properties"*, to allow a server to respond to the client that sent a message. A server sets up a service queue, with a name known to clients. A client creates a private queue for the server's response, creates a message for a request, sets the request's reply-to property to the address of the client's response queue, and sends the request to the service queue. The server sends the response to the address specified in the request's reply-to property.

Example 2.18. Request/Response Applications in C++

This example shows the C++ code for a client and server that use the request/response pattern.

The server creates a service queue and waits for a message to arrive. If it receives a message, it sends a message back to the sender.

```
Receiver receiver = session.createReceiver("service_queue; {create: always}");

Message request = receiver.fetch();
const Address& address = request.getReplyTo(); // Get "reply-to" from request ...
if (address) {
  Sender sender = session.createSender(address); // ... send response to "reply-to"
  Message response("pong!");
  sender.send(response);
  session.acknowledge();
}
```

The client creates a sender for the service queue, and also creates a response queue that is deleted when the client closes the receiver for the response queue. In the C++ client, if the address starts with the character *#*, it is given a unique name.

```
Sender sender = session.createSender("service_queue");

Address responseQueue("#response-queue; {create:always, delete:always}");
Receiver receiver = session.createReceiver(responseQueue);
```

```
Message request;
request.setReplyTo(responseQueue);
request.setContent("ping");
sender.send(request);
Message response = receiver.fetch();
std::cout << request.getContent() << " -> " << response.getContent() << std::endl;
```

The client sends the string *`ping`* to the server. The server sends the response *`pong`* back to the same client, using the *`replyTo`* property.

2.13. Performance Tips

- Consider prefetching messages for receivers (see *Section 2.6, "Receiver Capacity (Prefetch)"*). This helps eliminate roundtrips and increases throughput. Prefetch is disabled by default, and enabling it is the most effective means of improving throughput of received messages.
- Send messages asynchronously. Again, this helps eliminate roundtrips and increases throughput. The C++ and .NET clients send asynchronously by default, however the python client defaults to synchronous sends.
- Acknowledge messages in batches (see *Section 2.7, "Acknowledging Received Messages"*). Rather than acknowledging each message individually, consider issuing acknowledgments after n messages and/or after a particular duration has elapsed.
- Tune the sender capacity (see *Section 2.5, "Sender Capacity and Replay"*). If the capacity is too low the sender may block waiting for the broker to confirm receipt of messages, before it can free up more capacity.
- If you are setting a reply-to address on messages being sent by the c++ client, make sure the address type is set to either queue or topic as appropriate. This avoids the client having to determine which type of node is being referred to, which is required when handling reply-to in AMQP 0-10.
- For latency-sensitive applications, setting *`tcp-nodelay`* on **`qpidd`** and on client connections can help reduce the latency.

2.14. Cluster Failover

The messaging broker can be run in clustering mode, which provides high reliability through replicating state between brokers in the cluster. If one broker in a cluster fails, clients can choose another broker in the cluster and continue their work. Each broker in the cluster also advertises the addresses of all known brokers. [11] A client can use this information to dynamically keep the list of reconnection URLs up to date.

In C++, the **`FailoverUpdates`** class provides this functionality:

Example 2.19. Tracking cluster membership

In C++:

```
#include <qpid/messaging/FailoverUpdates.h>
```

[11] This is done via the **`amq.failover`** exchange in AMQP 0-10

```
...
Connection connection("localhost:5672");
connection.setOption("reconnect", true);
try {
    connection.open();
    std::auto_ptr<FailoverUpdates> updates(new FailoverUpdates(connection));
```

In Python:

```
import qpid.messaging.util
...
connection = Connection("localhost:5672")
connection.reconnect = True
try:
  connection.open()
  auto_fetch_reconnect_urls(connection)
```

In .NET C#:

```
using Org.Apache.Qpid.Messaging;
...
connection = new Connection("localhost:5672");
connection.SetOption("reconnect", true);
try {
  connection.Open();
  FailoverUpdates failover = new FailoverUpdates(connection);
```

2.15. Logging

To simplify debugging, Qpid provides a logging facility that prints out messaging events.

2.15.1. Logging in C++

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

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+"
```

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

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

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

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

2.15.2. Logging in Python

The 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()
```

The **`qpidd`** daemon 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.messaging.io", DEBUG)
```

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

2.16. Security

Qpid provides authentication, rule-based authorization, encryption, and digital signing.

Authentication is done using Simple Authentication and Security Layer (SASL) to authenticate client connections to the broker. SASL is a framework that supports a variety of authentication methods. For secure applications, we suggest CRAM-MD5, DIGEST-MD5, or GSSAPI (Kerberos). The ANONYMOUS method is not secure. The PLAIN method is secure only when used together with SSL.

To enable Kerberos in a client, set the `sasl_mechanisms` connection option to *`GSSAPI`*:

```

Connection connection(broker);
connection.setOption("sasl_mechanisms", "GSSAPI");
try {
    connection.open();
    ...
```

For Kerberos authentication, if the user running the program is already authenticated, for example, if they are using **`kinit`**, there is no need to supply a user name or password. If you are using another form of authentication, or are not already authenticated with Kerberos, you can supply these as connection options:

```

connection.setOption("username", "mick");
connection.setOption("password", "pa$$word");
```

Encryption and signing are done using SSL (they can also be done using SASL). To enable SSL, set the `protocol` connection option to *`ssl`*:

```

connection.setOption("protocol", "ssl");
```

Use the following environment variables to configure the SSL client:

Table 2.9. SSL Client Environment Variables for C++ clients

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

2.17. The AMQP 0-10 mapping

This section describes the AMQP 0-10 mapping for the Qpid Messaging API.

The interaction with the broker triggered by creating a sender or receiver depends on what the specified address resolves to. Where the node type is not specified in the address, the client queries the broker to determine whether it refers to a queue or an exchange.

When sending to a queue, the queue's name is set as the routing key and the message is transferred to the default (or nameless) exchange. When sending to an exchange, the message is transferred to that exchange and the routing key is set to the message subject if one is specified. A default subject may be specified in the target address. The subject may also be set on each message individually to override the default if required. In each case any specified subject is also added as a **`qpid.subject`** entry in the *`application-headers`* field of the *`message-properties`*.

When receiving from a queue, any subject in the source address is currently ignored. The client sends a *`message-subscribe`* request for the queue in question. The *`accept-mode`* is determined by the reliability option in the link properties; for unreliable links the *`accept-mode`* is none, for reliable links it is explicit. The default for a queue is reliable. The *`acquire-mode`* is determined by the value of the mode option. If the mode is set to browse the acquire mode is *`not-acquired`*, otherwise it is set to *`pre-acquired`*. The exclusive and arguments fields in the *`message-subscribe`* command can be controlled using the *`x-subscribe`* map.

When receiving from an exchange, the client creates a subscription queue and binds that to the exchange. The subscription queue's arguments can be specified using the *`x-declare`* map within the link properties. The reliability option determines most of the other parameters. If the reliability is set to *`unreliable`* then an auto-deleted, exclusive queue is used meaning that if the client or connection fails messages may be lost. For *`exactly-once`* the queue is not set to be auto-deleted. The durability of the subscription queue is determined by the durable option in the link properties. The binding process depends on the type of the exchange the source address resolves to.

- For a topic exchange, if no subject is specified and no *`x-bindings`* are defined for the link, the subscription queue is bound using a wildcard matching any routing key (thus satisfying the expectation that any message sent to that address will be received from it). If a subject is specified in the source address however, it is used for the binding key (this means that the subject in the source address may be a binding pattern including wildcards).

- For a fanout exchange the binding key is irrelevant to matching. A receiver created from a source address that resolves to a fanout exchange receives all messages sent to that exchange regardless of any subject the source address may contain. An *`x-bindings`* element in the link properties should be used if there is any need to set the arguments to the bind.

- For a direct exchange, the subject is used as the binding key. If no subject is specified an empty string is used as the binding key.

- For a headers exchange, if no subject is specified the binding arguments simply contain an *`x-match`* entry and no other entries, causing all messages to match. If a subject is specified then the binding arguments contain an *`x-match`* entry set to all and an entry for **`qpid.subject`** whose value is the subject in the source address (this means the subject in the source address must match the message subject exactly). For more control the *`x-bindings`* element in the link properties must be used.

- For the XML exchange,[12] if a subject is specified it is used as the binding key and an XQuery is defined that matches any message with that value for **`qpid.subject`**. Again this means that only messages whose subject exactly match that specified in the source address are received. If no subject is specified then the empty string is used as the binding key with an xquery that will match any message (this means that only messages with an empty string as the routing key will be received). For more control the x-bindings element in the link properties must be used. A source address that resolves to the XML exchange must contain either a subject or an x-bindings element in the link properties as there is no way at present to receive any message regardless of routing key.

If an x-bindings list is present in the link options a binding is created for each element within that list. Each element is a nested map that may contain values named *`queue`*, *`exchange`*, *`key`*, or *`arguments`*. If the queue value is absent the queue name the address resolves to is implied. If the exchange value is absent the exchange name the address resolves to is implied.

The following table shows how Qpid Messaging API message properties are mapped to AMQP 0-10 message properties and delivery properties. In this table `msg` refers to the Message class defined in the Qpid Messaging API, `mp` refers to an AMQP 0-10 `message-properties` struct, and `dp` refers to an AMQP 0-10 `delivery-properties` struct.

Table 2.10. Mapping to AMQP 0-10 Message Properties

Python API	C++ API[1]	AMQP 0-10 Property[2]
`msg.id`	**`msg.{get,set}MessageId()`**	**`mp.message_id`**
`msg.subject`	**`msg.{get,set}Subject()`**	**`mp.application_headers["qpid.subject"]`**
`msg.user_id`	**`msg.{get,set}UserId()`**	**`mp.user_id`**
`msg.reply_to`	**`msg.{get,set}ReplyTo()`**	**`mp.reply_to`**[3]
`msg.correlation_id`	**`msg.{get,set}CorrelationId()`**	**`mp.correlation_id`**
`msg.durable`	**`msg.{get,set}Durable()`**	**`dp.delivery_mode == delivery_mode.persistent`**[4]
`msg.priority`	**`msg.{get,set}Priority()`**	**`dp.priority`**
`msg.ttl`	**`msg.{get,set}Ttl()`**	**`dp.ttl`**
`msg.redelivered`	**`msg.{get,set}Redelivered()`**	**`dp.redelivered`**
`msg.properties`	**`msg.{get,set}Properties()`**	**`mp.application_headers`**
`msg.content_type`	**`msg.{get,set}ContentType()`**	**`mp.content_type`**

[1] The .NET Binding for C++ Messaging provides all the message and delivery properties described in the C++ API. See *Table 4.12, ".NET Binding for the C++ Messaging API Class: Message"* .

[2] In these entries, *mp* refers to an AMQP message property, and *dp* refers to an AMQP delivery property.

[3] The reply_to is converted from the protocol representation into an address.

[4] Note that msg.durable is a boolean, not an enum.

The 0-10 mapping also recognizes certain special property keys. If the properties contain entries for *`x-amqp-0-10.app-id`* or *`x-amqp-0-10.content-encoding`*, the values will be used to set *`message-properties.app-id`* and *`message-properties.content-encoding`* on the resulting 0-10 message transfer. Likewise if an incoming transfer has those properties set, they will be exposed in the same manner. In addition the routing key on incoming transfers will be exposed directly via the custom property with key *`x-amqp-0-10.routing-key`*.

2.18. Broker Exchange and Queue Configuration via QMF

The Qpid broker is managed by specially formatted messages sent to- and received from- special addresses. These messages can list, create and delete queues and exchanges, and bind them together. This approach to broker management forms part of the Qpid Management Framework (QMF) version 2.

Command messages are map messages that are sent to the address *`qmf.default.direct/broker`* where *`qmf.default.direct`* is the exchange, with a routing key or subject of *`broker`*. The message should contain a reply-to address from which the sender can receive responses.

2.18.1. Map Message Structure

The map message for commands follows a particular pattern. Within a map message there are entries containing the keys *`_object_id`*, *`_method_name`* and *`_arguments`*.

There must always be an entry with the key *`_object_id`* whose value is a nested map identifying the target of the command. Commands listed in the following section are specifically targeting the broker. For this reason, the *`_object_id`* map contains a single value with they key *`_object_name`* containing the value *`org.apache.qpid.broker:broker:amqp-broker`*. The key *`_method_name`* has the name of the command as its value and the key *`_arguments`* contains a nested map where the arguments for the command are.

In addition to the the correctly formatted content. Two message properties, *`x-amqp-0-10.app-id`* and *`qmf.opcode`* must be set. The property *`x-amqp-0-10.app-id`* should always have the value *`qmf2`* and *`qmf.opcode`* contains the value *`_method_request`*.

After the correctly constructed command message is sent to the correct address, you can wait for the response to arrive from the reply-to address specified. After the response arrives the *`x-amqp-0-10.app-id`* property should contain the value *`qmf2`*. The *`qmf.opcode`* property will contain the value *`_method_response`* if the message was processed as expected. If an error was encountered *`qmf.opcode`* property will contain the value *`_exception`*. In both cases the response content is again a map. In the case of a valid response, return values will be present as a nested map against the key *`_arguments`*. In the case of an exception, details of the exception will be within a nested map against the key *`_values`*.

2.18.2. Create and Delete Commands

The commands to create and delete a queue, exchange or binding between them are named **`create`** and **`delete`** respectively.

The **`create`** command takes four arguments:

- The *`type`* of object to be created, this can be a queue, exchange or binding.
- The *`name`* of the object to be created.
- The specific *`properties`* for the object to be created, value is a nested map.

- The *strict* argument takes a boolean value that is presently ignored. This value is intended to indicate whether the command will fail if any unrecognized properties have been specified.

The **create** command can also contain the argument *auto_delete_timeout* which if specified upon first declaring an auto-delete queue will allow you to specify a delay, in seconds, after which the deletion will take place. If the queue is re-declared after becoming eligible for deletion, but before the delay expires, then the queue will be not be deleted.

The **delete** command takes three arguments:

- The *type* of object to be deleted, acceptable values are queue, exchange or binding.
- The *name* to identify which object to delete.
- The last argument is a nested map with key *options*, presently unused.

The *name* argument of a queue or exchange is a single value, for example a queue named *my-queue* sets the *name* argument to a string of that value. The name of a binding uses the pattern *exchange/queue/key*, for example *amq.topic/my-queue/my-key* identifies a binding between *my-queue* and the exchange *amq.topic* with the binding key *my-key*.

2.18.3. Queue Creation

The following Python code example shows the creation of a queue named *my-queue*. In this example *my-queue* is configured to be auto-deleted after 10 seconds.

```
conn = Connection(opts.broker)
try:
  conn.open()
  ssn = conn.session()
  snd = ssn.sender("qmf.default.direct/broker")
  reply_to = "reply-queue; {create:always, node:{x-declare:{auto-delete:true}}}"
  rcv = ssn.receiver(reply_to)

  content = {
             "_object_id": {"_object_name": "org.apache.qpid.broker:broker:amqp-broker"},
             "_method_name": "create",
             "_arguments": {"type":"queue", "name":"my-queue", "properties":{"auto-
delete":True, "qpid.auto_delete_timeout":10}}
            }
  request = Message(reply_to=reply_to, content=content)
  request.properties["x-amqp-0-10.app-id"] = "qmf2"
  request.properties["qmf.opcode"] = "_method_request"
  snd.send(request)

  try:
    response = rcv.fetch(timeout=opts.timeout)
    if response.properties['x-amqp-0-10.app-id'] == 'qmf2':
      if response.properties['qmf.opcode'] == '_method_response':
        return response.content['_arguments']
      elif response.properties['qmf.opcode'] == '_exception':
        raise Exception("Error: %s" % response.content['_values'])
      else: raise Exception("Invalid response received, unexpected opcode: %s" % m)
    else: raise Exception("Invalid response received, not a qmfv2 method: %s" % m)
  except Empty:
    print "No response received!"
  except Exception, e:
    print e
except ReceiverError, e:
  print e
except KeyboardInterrupt:
  pass

conn.close()
```

Using the Qpid JMS client

3.1. A Simple Messaging Program in Java JMS

The following program shows how to send and receive a message using the Qpid JMS client. JMS programs typically use JNDI to obtain connection factory and destination objects which the application needs. In this way the configuration is kept separate from the application code itself.

This example shows how to create a JNDI context using a properties file, use the context to lookup a connection factory, create and start a connection, create a session, and look up a destination from the JNDI context. It then shows how to create a producer and a consumer, send a message with the producer and receive it with the consumer.

Example 3.1. "Hello world!" in Java

```
package org.apache.qpid.example.jmsexample.hello;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;

public class Hello {

  public Hello() {
  }

  public static void main(String[] args) {
    Hello producer = new Hello();
    producer.runTest();
  }

  private void runTest() {
    try {
      Properties properties = new Properties();
      properties.load(this.getClass().getResourceAsStream("hello.properties"));  «1»
      Context context = new InitialContext(properties);   «2»

      ConnectionFactory connectionFactory
          = (ConnectionFactory) context.lookup("qpidConnectionfactory"); «3»
      Connection connection = connectionFactory.createConnection();  «4»
      connection.start();  «5»

      Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE); «6»
      Destination destination = (Destination) context.lookup("topicExchange");  «7»

      MessageProducer messageProducer = session.createProducer(destination);  «8»
      MessageConsumer messageConsumer = session.createConsumer(destination);  «9»

      TextMessage message = session.createTextMessage("Hello world!");
      messageProducer.send(message);

      message = (TextMessage)messageConsumer.receive();    «10»
      System.out.println(message.getText());

      connection.close();  «11»
      context.close();   «12»
    }
    catch (Exception exp) {
      exp.printStackTrace();
    }
  }
}
```

«1» Loads the JNDI properties file, which specifies connection properties, queues, topics, and addressing options. See *Section 3.2, “Apache Qpid JNDI Properties for AMQP Messaging”* for details.

«2» Creates the JNDI initial context.

«3» Creates a JMS connection factory for Qpid.

«4» Creates a JMS connection.

«5» Activates the connection.

«6» Creates a session. This session is not transactional (transactions='false'), and messages are automatically acknowledged.

«7» Creates a destination for the topic exchange, so senders and receivers can use it.

«8» Creates a producer that sends messages to the topic exchange.

«9» Creates a consumer that reads messages from the topic exchange.

«10» Reads the next available message.

«11» Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

«12» Closes the JNDI context.

Example 3.2. JNDI Properties File for "Hello world!" Example

The contents of the **`hello.properties`** file:

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

# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory
  = amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'   «1»
# destination.[jndiname] = [address_string]
destination.topicExchange = amq.topic     «2»
```

«1» Defines a connection factory from which connections can be created. The syntax of a **`ConnectionURL`** is given in *Section 3.2, “Apache Qpid JNDI Properties for AMQP Messaging”*.

«2» Defines a destination for which **`MessageProducers`** and **`MessageConsumers`** can be created to send and receive messages. The value for the destination in the properties file is an address string as described in *Section 2.4, “Addresses”*. In the JMS implementation **`MessageProducers`** are analogous to senders in the Qpid Message API, and **`MessageConsumers`** are analogous to receivers.

3.2. Apache Qpid JNDI Properties for AMQP Messaging

Example 3.3. JNDI Properties File

Apache Qpid defines JNDI properties that can be used to specify JMS Connections and Destinations. This is a typical JNDI properties file:

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

# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory
  = amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'
# destination.[jndiname] = [address_string]
destination.topicExchange = amq.topic
```

3.2.1. JNDI Properties for Apache Qpid

Apache Qpid supports the properties shown in the following table:

Table 3.1. JNDI Properties supported by Apache Qpid

Property	Purpose
connectionfactory.<jndiname>	The Connection URL that the connection factory uses to perform connections.
queue.<jndiname>	A JMS queue, which is implemented as an amq.direct exchange in Apache Qpid.
topic.<jndiname>	A JMS topic, which is implemented as an amq.topic exchange in Apache Qpid.
destination.<jndiname>	Can be used for defining all amq destinations, queues, topics and header matching, using an address string. [1]

[1] Binding URLs, which were used in earlier versions of the Qpid Java JMS client, can still be used instead of address strings.

3.2.2. 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'
```

Apache Qpid supports the following properties in Connection URLs:

Table 3.2. Connection URL Properties

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.

Option	Type	Description
sync_publish	{'persistent' | 'all'}	A sync command is sent after every persistent message to guarantee that it has been received; if the value is 'persistent', this is done only for persistent messages.
sync_ack	Boolean	A sync command is sent after every acknowledgment to guarantee that it has been received.
use_legacy_map_msg_format	Boolean	If you are using JMS Map messages and deploying a new client with any JMS client older than 0.7 release, you must set this to true to ensure the older clients can understand the map message encoding.
failover	{'roundrobin' | 'failover_exchange'}	If roundrobin is selected it will try each broker given in the broker list. If failover_exchange is selected it connects to the initial broker given in the broker URL and will receive membership updates via the failover exchange.

Broker lists are specified using a URL in this format:

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

For instance, this is a typical broker list:

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

A broker list can contain more than one broker address; if so, the connection is made to the first broker in the list that is available. In general, it is better to use the failover exchange when using multiple brokers, since it allows applications to fail over if a broker goes down.

Example 3.4. Broker Lists

A broker list can specify properties to be used when connecting to the broker, such as security options. This broker list specifies options for a Kerberos connection using GSSAPI:

```
amqp://guest:guest@test/test?sync_ack='true'
    &brokerlist='tcp://ip1:5672?sasl_mechs='GSSAPI'
```

This broker list specifies SSL options:

```
amqp://guest:guest@test/test?sync_ack='true'
```

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

The following broker list options are supported.

Table 3.3. Broker List Options

Option	Type	Description
heartbeat	integer	frequency of heartbeat messages (in seconds)
sasl_mechs	--	For secure applications, we suggest CRAM-MD5, DIGEST-MD5, or GSSAPI. The ANONYMOUS method is not secure. The PLAIN method is secure only when used together with SSL. For Kerberos, sasl_mechs must be set to GSSAPI, sasl_protocol must be set to the principal for the qpidd broker, e.g. qpidd/, and sasl_server must be set to the host for the SASL server, e.g. sasl.com. SASL External is supported using SSL certification, e.g. **ssl='true'&sasl_mechs='EXTERNAL'**
sasl_encryption	Boolean	If **sasl_encryption='true'**, the JMS client attempts to negotiate a security layer with the broker using GSSAPI to encrypt the connection. Note that for this to happen, GSSAPI must be selected as the sasl_mech.
ssl	Boolean	If **ssl='true'**, the JMS client will encrypt the connection using SSL.
tcp_nodelay	Boolean	If **tcp_nodelay='true'**, TCP packet batching is disabled.
sasl_protocol	--	Used only for Kerberos. **sasl_protocol** must be set to the principal for the qpidd broker, e.g. **qpidd/**
sasl_server	--	For Kerberos, sasl_mechs must be set to GSSAPI, sasl_server must be set to the host for the SASL server, e.g. **sasl.com**.
trust_store	--	path to Kerberos trust store

Option	Type	Description
trust_store_password		Kerberos trust store password
key_store		path to Kerberos key store
key_store_password	--	Kerberos key store password
ssl_verify_hostname	Boolean	When using SSL you can enable hostname verification by using "ssl_verify_hostname=true" in the broker URL.
ssl_cert_alias		If multiple certificates are present in the keystore, the alias will be used to extract the correct certificate.

3.3. Java JMS Message Properties

The following table shows how Qpid Messaging API message properties are mapped to AMQP 0-10 message properties and delivery properties. In this table *msg* refers to the Message class defined in the Qpid Messaging API, *mp* refers to an AMQP 0-10 *message-properties* struct, and *dp* refers to an AMQP 0-10 *delivery-properties* struct.

Table 3.4. Java JMS Mapping to AMQP 0-10 Message Properties

Java JMS Message Property	AMQP 0-10 Property[1]
JMSMessageID	*mp.message_id*
qpid.subject[2]	*mp.application_headers["qpid.subject"]*
JMSXUserID	*mp.user_id*
JMSReplyTo	*mp.reply_to*[3]
JMSCorrelationID	*mp.correlation_id*
JMSDeliveryMode	*dp.delivery_mode*
JMSPriority	*dp.priority*
JMSExpiration	*dp.ttl*[4]
JMSRedelivered	*dp.redelivered*
JMS Properties	*mp.application_headers*
JMSType	*mp.content_type*

[1] In these entries, *mp* refers to an AMQP message property, and *dp* refers to an AMQP delivery property.

[2] This is a custom JMS property, set automatically by the Java JMS client implementation.

[3] The reply_to is converted from the protocol representation into an address.

[4] JMSExpiration = dp.ttl + currentTime

3.4. JMS MapMessage Types

Qpid supports the Java JMS **MapMessage** interface, which provides support for maps in messages. The following code shows how to send a **MapMessage** in Java JMS.

Example 3.5. Sending a Java JMS MapMessage

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Session;

import org.apache.qpid.client.AMQAnyDestination;
import org.apache.qpid.client.AMQConnection;

import edu.emory.mathcs.backport.java.util.Arrays;

// !!! SNIP !!!

MessageProducer producer = session.createProducer(queue);

MapMessage m = session.createMapMessage();
m.setIntProperty("Id", 987654321);
m.setStringProperty("name", "Widget");
m.setDoubleProperty("price", 0.99);

List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("white");
m.setObject("colours", colors);

Map<String,Double> dimensions = new HashMap<String,Double>();
dimensions.put("length",10.2);
dimensions.put("width",5.1);
dimensions.put("depth",2.0);
m.setObject("dimensions",dimensions);

List<List<Integer>> parts = new ArrayList<List<Integer>>();
parts.add(Arrays.asList(new Integer[] {1,2,5}));
parts.add(Arrays.asList(new Integer[] {8,2,5}));
m.setObject("parts", parts);

Map<String,Object> specs = new HashMap<String,Object>();
specs.put("colours", colors);
specs.put("dimensions", dimensions);
specs.put("parts", parts);
m.setObject("specs",specs);

producer.send(m);
```

The following table shows the data types that can be sent in a **MapMessage**, and the corresponding data types that will be received by clients in Python or C++.

Table 3.5. Java Data Types in Maps

Java Data Type	Python	C++
boolean	bool	bool
short	int | long	int16
int	int | long	int32
long	int | long	int64
float	float	float
double	float	double

Java Data Type	Python	C++
`java.lang.String`	`unicode`	`std::string`
`java.util.UUID`	`uuid`	`qpid::types::Uuid`
`java.util.Map`[1]	`dict`	`Variant::Map`
`java.util.List`	`list`	`Variant::List`

[1] In Qpid, maps can nest. This goes beyond the functionality required by the JMS specification.

3.5. JMS Client Logging

The JMS Client logging is handled using the Simple Logging Facade for Java (*SLF4J*[1]). As the name implies, SLF4J is a facade that delegates to other logging systems like log4j or JDK 1.4 logging. For more information on how to configure SLF4J for specific logging systems, please consult the SLF4J documentation.

When using the log4j binding, please set the log level for **`org.apache.qpid`** explicitly. Otherwise log4j will default to *`DEBUG`* which will degrade performance considerably due to excessive logging. The recommended logging level for production is *`WARN`*.

The following example shows the logging properties used to configure client logging for slf4j using the log4j binding. These properties can be placed in a **`log4j.properties`** file and placed in the **`CLASSPATH`**, or they can be set explicitly using the *`-Dlog4j.configuration`* property.

Example 3.6. log4j Logging Properties

```
log4j.logger.org.apache.qpid=WARN, console
log4j.additivity.org.apache.qpid=false

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Threshold=all
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%t %d %p [%c{4}] %m%n
```

[1] http://www.slf4j.org/

.NET Binding for Qpid C++ Messaging

The .NET Binding for the C++ Qpid Messaging Client is a library that gives any .NET program access to Qpid C++ Messaging objects and methods.

Note

The .NET messaging managed callback library is a managed .NET assembly built using the .NET Framework v2.0. This component will run on any system with .NET Framework v2.0 or higher. None of the other components in the .NET binding has any dependency on a .NET Framework version.

4.1. .NET Binding for the C++ Messaging Client Examples

This chapter describes the various sample programs that are available to illustrate common Qpid Messaging usage.

Table 4.1. Client and Server Examples

Example Name	Example Description
`csharp.example.server`	Creates a receiver and listens for messages. Upon receipt, the content of the message is converted to upper case and forwarded to the received message's ReplyTo address.
`csharp.example.client`	Sends a series of messages to the server and prints the original message content and the received message content.

Table 4.2. Map Sender and Receiver Examples

Example Name	Example Description
`csharp.map.receiver`	Creates a receiver and listens for a map message. Upon receipt, the message is decoded and displayed on the console.
`csharp.map.sender`	Creates a map message and sends it to `map.receiver`. The map message contains values for every supported .NET messaging binding data type.

Table 4.3. Spout and Drain Examples

Example Name	Example Description
`csharp.example.spout`	`Spout` is a more complex example of code that generates a series of messages and sends them to the `Drain` peer program. Flexible command line arguments allow the user to specify a variety of message and program options.
`csharp.example.drain`	`Drain` is a more complex example of code that receives a series of messages and displays their contents on the console.

Table 4.4. Map Callback Sender and Receiver Examples

Example Name	Example Description
`csharp.map.callback.receiver`	Creates a receiver and listens for a map message. Upon message reception the message is decoded and displayed on the console. This example illustrates the use of the C# managed code callback mechanism provided by the .NET messaging binding managed callback library.
`csharp.map.callback.sender`	Creates a map message and sends it to `map_receiver`. The map message contains values for every supported .NET messaging binding data type.

Table 4.5. Declare Queues Examples

Example Name	Example Description
`csharp.example.declare_queues`	A program to illustrate creating objects on a broker. This program creates a queue used by `Spout` and `Drain`.

Table 4.6. Direct Sender and Receiver Examples

Example Name	Example Description
`csharp.direct.receiver`	Creates a receiver and listens for a messages. Upon receipt, the message is decoded and displayed on the console.
`csharp.direct.sender`	Creates a series of messages and sends them to `csharp.direct.receiver`.

Table 4.7. Hello World Example

Example Name	Example Description
`csharp.example.helloworld`	A program to send a message and to receive the same message.

4.2. .NET Binding Class Mapping to Underlying C++ Messaging API

This chapter describes the specific mappings between classes in the .NET binding and the underlying C++ messaging API.

4.2.1. .NET Binding for the C++ Messaging API Class: Address

Table 4.8. .NET Binding for the C++ Messaging API Class: Address

.NET Binding Class: Address	
Language	Syntax
C++	*class Address*
.NET	*public ref class Address*
Constructor	
C++	*Address();*
.NET	*public Address();*

.NET Binding Class: Address	
Language	Syntax
Constructor	
C++	*Address(const std::string& address);*
.NET	*public Address(string address);*
Constructor	
C++	*Address(const std::string& name, const std::string& subject, const qpid::types::Variant::Map& options, const std::string& type = "");*
.NET	*public Address(string name, string subject, Dictionary<string, object> options);*
.NET	*public Address(string name, string subject, Dictionary<string, object> options, string type);*
Copy constructor	
C++	*Address(const Address& address);*
.NET	*public Address(Address address);*
Destructor	
C++	*~Address();*
.NET	*~Address();*
Finalizer	
C++	not applicable
.NET	*!Address();*
Copy assignment operator	
C++	*Address& operator=(const Address&);*
.NET	*public Address op_Assign(Address rhs);*
Property: Name	
C++	*const std::string& getName() const;*
C++	*void setName(const std::string&);*
.NET	*public string Name { get; set; }*
Property: Subject	
C++	*const std::string& getSubject() const;*
C++	*void setSubject(const std::string&);*
.NET	*public string Subject { get; set; }*
Property: Options	
C++	*const qpid::types::Variant::Map& getOptions() const;*
C++	*qpid::types::Variant::Map& getOptions();*
C++	*void setOptions(const qpid::types::Variant::Map&);*
.NET	*public Dictionary<string, object> Options { get; set; }*
Property: Type	
C++	*std::string getType() const;*

.NET Binding Class: Address	
Language	**Syntax**
C++	*void setType(const std::string&);*
.NET	*public string Type { get; set; }*
Miscellaneous	
C++	*std::string str() const;*
.NET	*public string ToStr();*
Miscellaneous	
C++	*operator bool() const;*
.NET	not applicable
Miscellaneous	
C++	*bool operator !() const;*
.NET	not applicable

4.2.2. .NET Binding for the C++ Messaging API Class: Connection

Table 4.9. .NET Binding for the C++ Messaging API Class: Connection

.NET Binding Class: Connection	
Language	**Syntax**
C++	*class Connection : public qpid::messaging::Handle<ConnectionImpl>*
.NET	*public ref class Connection*
Constructor	
C++	*Connection(ConnectionImpl* impl);*
.NET	not applicable
Constructor	
C++	*Connection();*
.NET	not applicable
Constructor	
C++	*Connection(const std::string& url, const qpid::types::Variant::Map& options = qpid::types::Variant::Map());*
.NET	*public Connection(string url);*
.NET	*public Connection(string url, Dictionary<string, object> options);*
Constructor	
C++	*Connection(const std::string& url, const std::string& options);*
.NET	*public Connection(string url, string options);*
Copy Constructor	
C++	*Connection(const Connection&);*
.NET	*public Connection(Connection connection);*
Destructor	

.NET Binding Class: Connection	
Language	**Syntax**
C++	*~Connection();*
.NET	*~Connection();*
	Finalizer
C++	not applicable
.NET	*!Connection();*
	Copy assignment operator
C++	*Connection& operator=(const Connection&);*
.NET	*public Connection op_Assign(Connection rhs);*
	Method: SetOption
C++	*void setOption(const std::string& name, const qpid::types::Variant& value);*
.NET	*public void SetOption(string name, object value);*
	Method: open
C++	*void open();*
.NET	*public void Open();*
	Property: isOpen
C++	*bool isOpen();*
.NET	*public bool IsOpen { get; }*
	Method: close
C++	*void close();*
.NET	*public void Close();*
	Method: createTransactionalSession
C++	*Session createTransactionalSession(const std::string& name = std::string());*
.NET	*public Session CreateTransactionalSession();*
.NET	*public Session CreateTransactionalSession(string name);*
	Method: createSession
C++	*Session createSession(const std::string& name = std::string());*
.NET	*public Session CreateSession();*
.NET	*public Session CreateSession(string name);*
	Method: getSession
C++	*Session getSession(const std::string& name) const;*
.NET	*public Session GetSession(string name);*
	Property: AuthenticatedUsername
C++	*std::string getAuthenticatedUsername();*
.NET	*public string GetAuthenticatedUsername();*

4.2.3. .NET Binding for the C++ Messaging API Class: Duration

Table 4.10. .NET Binding for the C++ Messaging API Class: Duration

.NET Binding Class: Duration	
Language	**Syntax**
C++	*class Duration*
.NET	*public ref class Duration*
Constructor	
C++	*explicit Duration(uint64_t milliseconds);*
.NET	*public Duration(ulong mS);*
Copy constructor	
C++	not applicable
.NET	*public Duration(Duration rhs);*
Destructor	
C++	default
.NET	default
Finalizer	
C++	not applicable
.NET	default
Property: Milliseconds	
C++	*uint64_t getMilliseconds() const;*
.NET	*public ulong Milliseconds { get; }*
Operator: *	
C++	*Duration operator*(const Duration& duration, uint64_t multiplier);*
.NET	*public static Duration operator *(Duration dur, ulong multiplier);*
.NET	*public static Duration Multiply(Duration dur, ulong multiplier);*
C++	*Duration operator*(uint64_t multiplier, const Duration& duration);*
.NET	*public static Duration operator *(ulong multiplier, Duration dur);*
.NET	*public static Duration Multiply(ulong multiplier, Duration dur);*
Constants	
C++	*static const Duration FOREVER;*
C++	*static const Duration IMMEDIATE;*
C++	*static const Duration SECOND;*
C++	*static const Duration MINUTE;*
.NET	*public sealed class DurationConstants*
.NET	*public static Duration FORVER;*
.NET	*public static Duration IMMEDIATE;*

.NET Binding Class: Duration	
Language	**Syntax**
.NET	*public static Duration MINUTE;*
.NET	*public static Duration SECOND;*

4.2.4. .NET Binding for the C++ Messaging API Class: FailoverUpdates

Table 4.11. .NET Binding for the C++ Messaging API Class: FailoverUpdates

.NET Binding Class: FailoverUpdates	
Language	**Syntax**
C++	*class FailoverUpdates*
.NET	*public ref class FailoverUpdates*
Constructor	
C++	*FailoverUpdates(Connection& connection);*
.NET	*public FailoverUpdates(Connection connection);*
Destructor	
C++	*~FailoverUpdates();*
.NET	*~FailoverUpdates();*
Finalizer	
C++	not applicable
.NET	*!FailoverUpdates();*

4.2.5. .NET Binding for the C++ Messaging API Class: Message

Table 4.12. .NET Binding for the C++ Messaging API Class: Message

.NET Binding Class: Message	
Language	**Syntax**
C++	*class Message*
.NET	*public ref class Message*
Constructor	
C++	*Message(const std::string& bytes = std::string());*
.NET	*Message();*
.NET	*Message(System::String ^ theStr);*
.NET	*Message(System::Object ^ theValue);*
.NET	*Message(array<System::Byte> ^ bytes);*
Constructor	
C++	*Message(const char*, size_t);*
.NET	*public Message(byte[] bytes, int offset, int size);*
Copy Constructor	
C++	*Message(const Message&);*

.NET Binding Class: Message	
Language	**Syntax**
.NET	*public Message(Message message);*
Copy assignment operator	
C++	*Message& operator=(const Message&);*
.NET	*public Message op_Assign(Message rhs);*
Destructor	
C++	*~Message();*
.NET	*~Message();*
Finalizer	
C++	not applicable
.NET	*!Message()*
Property: ReplyTo	
C++	*void setReplyTo(const Address&);*
C++	*const Address& getReplyTo() const;*
.NET	*public Address ReplyTo { get; set; }*
Property: Subject	
C++	*void setSubject(const std::string&);*
C++	*const std::string& getSubject() const;*
.NET	*public string Subject { get; set; }*
Property: ContentType	
C++	*void setContentType(const std::string&);*
C++	*const std::string& getContentType() const;*
.NET	*public string ContentType { get; set; }*
Property: MessageId	
C++	*void setMessageId(const std::string&);*
C++	*const std::string& getMessageId() const;*
.NET	*public string MessageId { get; set; }*
Property: UserId	
C++	*void setUserId(const std::string&);*
C++	*const std::string& getUserId() const;*
.NET	*public string UserId { get; set; }*
Property: CorrelationId	
C++	*void setCorrelationId(const std::string&);*
C++	*const std::string& getCorrelationId() const;*
.NET	*public string CorrelationId { get; set; }*
Property: Priority	
C++	*void setPriority(uint8_t);*
C++	*uint8_t getPriority() const;*

.NET Binding Class: Message	
Language	**Syntax**
.NET	*public byte Priority { get; set; }*
	Property: Ttl
C++	*void setTtl(Duration ttl);*
C++	*Duration getTtl() const;*
.NET	*public Duration Ttl { get; set; }*
	Property: Durable
C++	*void setDurable(bool durable);*
C++	*bool getDurable() const;*
.NET	*public bool Durable { get; set; }*
	Property: Redelivered
C++	*bool getRedelivered() const;*
C++	*void setRedelivered(bool);*
.NET	*public bool Redelivered { get; set; }*
	Method: SetProperty
C++	*void setProperty(const std::string&, const qpid::types::Variant&);*
.NET	*public void SetProperty(string name, object value);*
	Property: Properties
C++	*const qpid::types::Variant::Map& getProperties() const;*
C++	*qpid::types::Variant::Map& getProperties();*
.NET	*public Dictionary<string, object> Properties { get; set; }*
	Method: SetContent
C++	*void setContent(const std::string&);*
C++	*void setContent(const char* chars, size_t count);*
.NET	*public void SetContent(byte[] bytes);*
.NET	*public void SetContent(string content);*
.NET	*public void SetContent(byte[] bytes, int offset, int size);*
	Method: GetContent
C++	*std::string getContent() const;*
.NET	*public string GetContent();*
.NET	*public void GetContent(byte[] arr);*
.NET	*public void GetContent(Collection<object> __p1);*
.NET	*public void GetContent(Dictionary<string, object> dict);*
	Method: GetContentPtr
C++	*const char* getContentPtr() const;*
.NET	not applicable
	Property: ContentSize
C++	*size_t getContentSize() const;*

.NET Binding Class: Message	
Language	**Syntax**
.NET	*public ulong ContentSize { get; }*
Struct: EncodingException	
C++	*struct EncodingException : qpid::types::Exception*
.NET	not applicable
Method: decode	
C++	*void decode(const Message& message, qpid::types::Variant::Map& map, const std::string& encoding = std::string());*
C++	*void decode(const Message& message, qpid::types::Variant::List& list, const std::string& encoding = std::string());*
.NET	not applicable
Method: encode	
C++	*void encode(const qpid::types::Variant::Map& map, Message& message, const std::string& encoding = std::string());*
C++	*void encode(const qpid::types::Variant::List& list, Message& message, const std::string& encoding = std::string());*
.NET	not applicable
Method: AsString	
C++	not applicable
.NET	*public string AsString(object obj);*
.NET	*public string ListAsString(Collection<object> list);*
.NET	*public string MapAsString(Dictionary<string, object> dict);*

4.2.6. .NET Binding for the C++ Messaging API Class: Receiver

Table 4.13. .NET Binding for the C++ Messaging API Class: Receiver

.NET Binding Class: Receiver	
Language	**Syntax**
C++	*class Receiver*
.NET	*public ref class Receiver*
Constructor	
.NET	*Constructed object is returned by Session.CreateReceiver*
Copy constructor	
C++	*Receiver(const Receiver&);*
.NET	*public Receiver(Receiver receiver);*
Destructor	
C++	*~Receiver();*
.NET	*~Receiver();*
Finalizer	
C++	not applicable

.NET Binding Class: Receiver	
Language	**Syntax**
.NET	*!Receiver()*
Copy assignment operator	
C++	*Receiver& operator=(const Receiver&);*
.NET	*public Receiver op_Assign(Receiver rhs);*
Method: Get	
C++	*bool get(Message& message, Duration timeout=Duration::FOREVER);*
.NET	*public bool Get(Message mmsgp);*
.NET	*public bool Get(Message mmsgp, Duration durationp);*
Method: Get	
C++	*Message get(Duration timeout=Duration::FOREVER);*
.NET	*public Message Get();*
.NET	*public Message Get(Duration durationp);*
Method: Fetch	
C++	*bool fetch(Message& message, Duration timeout=Duration::FOREVER);*
.NET	*public bool Fetch(Message mmsgp);*
.NET	*public bool Fetch(Message mmsgp, Duration duration);*
Method: Fetch	
C++	*Message fetch(Duration timeout=Duration::FOREVER);*
.NET	*public Message Fetch();*
.NET	*public Message Fetch(Duration durationp);*
Property: Capacity	
C++	*void setCapacity(uint32_t);*
C++	*uint32_t getCapacity();*
.NET	*public uint Capacity { get; set; }*
Property: Available	
C++	*uint32_t getAvailable();*
.NET	*public uint Available { get; }*
Property: Unsettled	
C++	*uint32_t getUnsettled();*
.NET	*public uint Unsettled { get; }*
Method: Close	
C++	*void close();*
.NET	*public void Close();*
Property: IsClosed	
C++	*bool isClosed() const;*
.NET	*public bool IsClosed { get; }*
Property: Name	

.NET Binding Class: Receiver	
Language	**Syntax**
C++	*const std::string& getName() const;*
.NET	*public string Name { get; }*
Property: Session	
C++	*Session getSession() const;*
.NET	*public Session Session { get; }*

4.2.7. .NET Binding for the C++ Messaging API Class: Sender

Table 4.14. .NET Binding for the C++ Messaging API Class: Sender

.NET Binding Class: Sender	
Language	**Syntax**
C++	*class Sender*
.NET	*public ref class Sender*
Constructor	
.NET	*Constructed object is returned by Session.CreateSender*
Copy constructor	
C++	*Sender(const Sender&);*
.NET	*public Sender(Sender sender);*
Destructor	
C++	*~Sender();*
.NET	*~Sender();*
Finalizer	
C++	not applicable
.NET	*!Sender()*
Copy assignment operator	
C++	*Sender& operator=(const Sender&);*
.NET	*public Sender op_Assign(Sender rhs);*
Method: Send	
C++	*void send(const Message& message, bool sync=false);*
.NET	*public void Send(Message mmsgp);*
.NET	*public void Send(Message mmsgp, bool sync);*
Method: Close	
C++	*void close();*
.NET	*public void Close();*
Property: Capacity	
C++	*void setCapacity(uint32_t);*
C++	*uint32_t getCapacity();*
.NET	*public uint Capacity { get; set; }*

.NET Binding Class: Sender	
Language	**Syntax**
Property: Available	
C++	*uint32_t getAvailable();*
.NET	*public uint Available { get; }*
Property: Unsettled	
C++	*uint32_t getUnsettled();*
.NET	*public uint Unsettled { get; }*
Property: Name	
C++	*const std::string& getName() const;*
.NET	*public string Name { get; }*
Property: Session	
C++	*Session getSession() const;*
.NET	*public Session Session { get; }*

4.2.8. .NET Binding for the C++ Messaging API Class: Session

Table 4.15. .NET Binding for the C++ Messaging API Class: Session

.NET Binding Class: Session	
Language	**Syntax**
C++	*class Session*
.NET	*public ref class Session*
Constructor	
.NET	Constructed object is returned by `Connection.CreateSession`
Copy constructor	
C++	*Session(const Session&);*
.NET	*public Session(Session session);*
Destructor	
C++	*~Session();*
.NET	*~Session();*
Finalizer	
C++	not applicable
.NET	*!Session()*
Copy assignment operator	
C++	*Session& operator=(const Session&);*
.NET	*public Session op_Assign(Session rhs);*
Method: Close	
C++	*void close();*
.NET	*public void Close();*
Method: Commit	

.NET Binding Class: Session	
Language	**Syntax**
C++	*void commit();*
.NET	*public void Commit();*
Method: Rollback	
C++	*void rollback();*
.NET	*public void Rollback();*
Method: Acknowledge	
C++	*void acknowledge(bool sync=false);*
C++	*void acknowledge(Message&, bool sync=false);*
.NET	*public void Acknowledge();*
.NET	*public void Acknowledge(bool sync);*
.NET	*public void Acknowledge(Message __p1);*
.NET	*public void Acknowledge(Message __p1, bool __p2);*
Method: Reject	
C++	*void reject(Message&);*
.NET	*public void Reject(Message __p1);*
Method: Release	
C++	*void release(Message&);*
.NET	*public void Release(Message __p1);*
Method: Sync	
C++	*void sync(bool block=true);*
.NET	*public void Sync();*
.NET	*public void Sync(bool block);*
Property: Receivable	
C++	*uint32_t getReceivable();*
.NET	*public uint Receivable { get; }*
Property: UnsettledAcks	
C++	*uint32_t getUnsettledAcks();*
.NET	*public uint UnsettledAcks { get; }*
Method: NextReceiver	
C++	*bool nextReceiver(Receiver&, Duration timeout=Duration::FOREVER);*
.NET	*public bool NextReceiver(Receiver rcvr);*
.NET	*public bool NextReceiver(Receiver rcvr, Duration timeout);*
Method: NextReceiver	
C++	*Receiver nextReceiver(Duration timeout=Duration::FOREVER);*
.NET	*public Receiver NextReceiver();*
.NET	*public Receiver NextReceiver(Duration timeout);*
Method: CreateSender	

.NET Binding Class: Session	
Language	**Syntax**
C++	*Sender createSender(const Address& address);*
.NET	*public Sender CreateSender(Address address);*
Method: CreateSender	
C++	*Sender createSender(const std::string& address);*
.NET	*public Sender CreateSender(string address);*
Method: CreateReceiver	
C++	*Receiver createReceiver(const Address& address);*
.NET	*public Receiver CreateReceiver(Address address);*
Method: CreateReceiver	
C++	*Receiver createReceiver(const std::string& address);*
.NET	*public Receiver CreateReceiver(string address);*
Method: GetSender	
C++	*Sender getSender(const std::string& name) const;*
.NET	*public Sender GetSender(string name);*
Method: GetReceiver	
C++	*Receiver getReceiver(const std::string& name) const;*
.NET	*public Receiver GetReceiver(string name);*
Property: Connection	
C++	*Connection getConnection() const;*
.NET	*public Connection Connection { get; }*
Property: HasError	
C++	*bool hasError();*
.NET	*public bool HasError { get; }*
Method: CheckError	
C++	*void checkError();*
.NET	*public void CheckError();*

4.2.9. .NET Class: SessionReceiver

The `SessionReceiver` class provides a convenient callback mechanism for messages received by all receivers on a given session.

```
using Org.Apache.Qpid.Messaging;
using System;

namespace Org.Apache.Qpid.Messaging.SessionReceiver
{
    public interface ISessionReceiver
    {
        void SessionReceiver(Receiver receiver, Message message);
    }

    public class CallbackServer
    {
        public CallbackServer(Session session, ISessionReceiver callback);

        public void Close();
    }
}
```

To use this class a client program includes references to both **`Org.Apache.Qpid.Messaging`** and **`Org.Apache.Qpid.Messaging.SessionReceiver`**. The calling program creates a function that implements the **`ISessionReceiver`** interface. This function will be called whenever a message is received by the session. The callback process is started by creating a **`CallbackServer`** and will continue to run until the client program calls the **`CallbackServer.Close function`**.

A complete operating example of using the **`SessionReceiver`** callback is contained in **`cpp/bindings/qpid/dotnet/examples/csharp.map.callback.receiver`**.

Appendix A. Revision History

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

Prepared for publishing

Revision 0.1-6 **Wed Jun 15 2011** **Alison Young** *`alyoung@redhat.com`*

BZ#651765 - sasl_mechanisms update

Revision 0.1-5 **Mon Jun 06 2011** **Alison Young** *`alyoung@redhat.com`*

BZ#651765 - sasl_mechanism fix

Revision 0.1-4 **Fri Jun 03 2011** **Alison Young** *`alyoung@redhat.com`*

Minor XML updates

Revision 0.1-3 **Thu May 19 2011** **Alison Young** *`alyoung@redhat.com`*

Technical review updates
BZ#683597 - using timed autodelete on queues
BZ#683600 - programatically configure exchanges, queues, and bindings from the new API

Revision 0.1-2 **Tue May 03 2011** **Alison Young** *`alyoung@redhat.com`*

BZ#683600 - programatically configure exchanges, queues, and bindings from the new API
BZ#693888 - Indexes in example code are reversed

Revision 0.1-1 **Thu Mar 31 2011** **Alison Young** *`alyoung@redhat.com`*

BZ#651765 - sasl-mechanism is not valid connection option for python client
Minor XML updates

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

Fork from 1.3