Red Hat Enterprise MRG 2 Messaging Programming Reference

A Guide to Programming with Red Enterprise Messaging

Joshua Wulf

Red Hat Enterprise MRG 2 Messaging Programming Reference

A Guide to Programming with Red Enterprise Messaging

Joshua Wulf
jwulf@redhat.com

Keywords

Abstract

This guide provides information for developers writing applications that utilize the Red Hat Enterprise Messaging Server

Table of Contents

Preface

1. Document Conventions

1.1. Default Programming Language for Code Samples

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

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

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

2. We Appreciate Your Feedback

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

Chapter 1. Introduction

1.1. Red Hat Enterprise MRG Messaging

Red Hat Enterprise Messaging is a highly scalable AMQP messaging broker and set of client libraries and tools based on the Apache Qpid open source project. It is integrated, tested, and supported by Red Hat for Enterprise customers.

Report a bug

1.2. Apache Qpid

Apache Qpid is a cross-platform Enterprise Messaging system that implements the Advanced Messaging Queue Protocol (AMQP). It is developed as an Apache Software Foundation open source project.

With Apache Qpid we strive to wrap an intuitive and easy to use messaging API around the AMQP model to handle as much of the complexity as possible (while still allowing you access to the nuts and bolts when you really need it), so that you can build highly performant and scalable applications with integrated messaging quickly and easily.

Report a bug

1.3. AMQP - Advanced Message Queuing Protocol

AMQP, the Advanced Message Queuing Protocol, is an open standard for interoperable messaging at the wire protocol level. Message brokers that implement AMQP can communicate with each other and exchange messages without the need for adapters or bridges. An AMQP message broker can provide first-class native language bindings for multiple programming languages; so AMQP-based messaging is a good choice for cross-platform compatibility across the Enterprise.

The AMQP standard is stewarded by a vendor-neutral OASIS Technical Committee.

Report a bug

1.4. AMQP 0-10

The version of AMQP supported by MRG Messaging 2.2 is AMQP 0-10.

Report a bug

1.5. Differences between AMQP 0-10 and AMQP 1.0

AMQP 1.0 is the latest standard for AMQP. The current version of Red Hat Enterprise Messaging supports AMQP 0-10, a previous version of the standard. Some of the most significant differences between AMQP 0-10 and AMQP 1.0 are described here to provide context to the AMQP model used in this product.

Broker Architecture

AMQP 0-10 provides a specification for the on-the-wire protocol *and* the broker architecture (in the form of exchange, bindings, and queues). AMQP 1.0, on the other hand, provides only a protocol specification, saying nothing about broker architecture. AMQP 1.0 does not require that there be a broker, exchanges, or bindings. It does not rules them out either.

The MRG-M broker is an "AMQP 0-10" broker that will, in the near future, provide protocol support for both AMQP 0-10 and AMQP 1.0.

Concepts such as "exchange" and "binding" are 0-10 concepts. They will, however continue to be used long after the 0-10 protocol is deprecated.

Broker Management

AMQP 0-10 defines protocol commands that are used to manage the broker. Examples include "Queue Declare", "Queue Delete", "Queue Query", etc. AMQP 1.0 does not include such commands and assumes that such capability will be added at a higher layer.

Note that the MRG-M broker also has a layered management capability (called Qpid Management Framework, QMF). It is expected that QMF will continue to be used in the future over AMQP 1.0.

Symmetry

The AMQP 0-10 protocol is asymmetric in that each connection is defined to have a "client" end and a "broker" end. As such, AMQP 0-10 is very broker-oriented.

AMQP 1.0 is symmetric and places no such constraints on the roles of connection endpoints. 1.0 permits brokerless point-to-point communication. It also permits the creation of servers/intermediaries that are not brokers in a strict sense.

Report a bug

Chapter 2. AMQP Model Overview

2.1. The Producer - Consumer Model

AMQP Messaging uses a Producer - Consumer model. Communication between the message producers and message consumers is decoupled by a broker that provides exchanges and queues. This allows applications to produce and consume data at different rates. Producers send messages to exchanges on the message broker. Consumers subscribe to exchanges that contain messages of interest, creating subscription queues that buffer messages for the consumer. Message producers can also create subscription queues and publish them for consuming applications.

The messaging broker functions as a decoupling layer, providing exchanges that distribute messages, the ability for consumers and producers to create public and private queues and subscribe them to exchanges, and buffering messages that are sent at-will by producer applications, and delivered on-demand to interested consumers.

Report a bug

2.2. Consumer-driven messaging

AMQP uses *consumer-driven* messaging. In traditional point-to-point messaging a message producer publishes messages to a queue. The message producer is responsible for knowing which queue will receive the messages. The queue in this model is an endpoint for a single consumer. In the traditional publish-subscribe model, the queue can be an endpoint for multiple consumers, who can receive individual copies of the messages sent to queue, or can share access to unique messages, taking them in a round-robin fashion. In AMQP all of these styles of messaging are supported: sending directly to a known queue for a single consumer or for multiple consumers, allowing consumers to browse their own copies of messages on the queue or mandating that they share access to unique instances of messages in a round-robin fashion.

AMQP implements these patterns using a flexible architecture where senders send their messages to an *exchange*. The exchange distributes the message to the queues subscribed to the exchange. This allows all the previously described models, and also provides the opportunity for message consumers to drive the conversation. Message producing applications do not need to be aware of new applications that come online and are interested in the message producer's messages. Message consumers can create queues and bind them to exchanges.

AMQP has a number of exchange types that support different distribution mechanisms. When subscribing to an exchange, message consumers can bind their queue with parameters that act as a filter on messages. By choosing which exchange type to use, and using binding keys to filter the messages from that exchange, you can build extremely flexible, fast, and extensible messaging systems using AMQP.

Report a bug

2.3. Message Producer (Sender)

Message producing applications send messages to an exchange on the message broker. The exchange then distributes the messages to the queues that are subscribed to the exchange. Depending on the type of exchange and the parameters used to subscribe the queue, messages are filtered so that each queue subscribed to the exchange gets only the messages that are of interest.

Message producers can send their messages with no knowledge of or interest in the consumers. Because they send to an exchange, they are decoupled from the receivers of the messages. Consumers can then control how and what messages they receive. Producers can also control how their messages are consumed by creating and subscribing a queue, and route the messages they send to the exchange to that queue. In this way a wide range of designs are possible.

Report a bug

2.4. Message

Applications produce information that is of interest to other applications. To share that information, they can create a portable unit that wraps the information and makes it transportable - a message.

A message consists of a *message content* - information of interest to a message receiving application; and *message headers*, information about the message itself, such as where it should be routed, how it should be treated while in transit, and what has happened to it during its transmission.

Report a bug

2.5. Message Broker

Messages can be sent directly between two applications, but this requires the two applications to know about each other when they are written; it also means that both applications need to be online at the same time and producing and consuming data at the same rate to communicate. This hard-wiring of communication between applications does not scale as more and more applications become interested in the information being shared.

A *message broker* provides a decoupling layer. By sending messages to a third party - the message broker - a message-producing application no longer has to know about all the applications that are interested in its information. The message broker can provide queues that carry the messages to interested message consuming applications. The message broker also provides a buffer that allows the applications involved to produce and consume data at different rates.

Red Hat Enterprise Messaging provides a messaging broker based on the Apache Qpid project. It implements AMQP (Advanced Messaging Queue Protocol) messaging.

Report a bug

2.6. Routing Key

The *Routing Key* is a string in the message that is used by the message broker to route the message for delivery. In Red Hat Enterprise Messaging, the *message subject* is used for routing.

Messages have an internal `x-ampq-0.10-routing-key` property. However, this is managed by the Qpid Messaging API, and you do not need to manually access or set this property. The exception to this is if you are exchanging messages with another AMQP system. In that case you should understand how the Qpid Messaging API manages this property based on message and sender subject.

- Section 19.3, “AMQP Routing Key and Message Subject”

Report a bug

2.7. Message Subject

A message has a subject property. This subject is used for message routing, and is synonomous with *routing key*.

Since the message subject is used for routing, it is not analogous to an email subject. In an email message the email address is used to route the email message to its recipient, and the email subject is available to describe the contents of the message. Since the message subject in a Qpid message is used to route a message, it is somewhat more like an email address, including the ability to send an email to one or multiple recipients with a single address.

A message's subject can be blank, it can be explicitly set manually, or it can be automatically set when the message is sent based on where it is to be routed.

Since the message subject can be automatically set when it is sent, you can develop applications where you never deal with the message subject, allowing it to be set by a sender. Or you can use a more generic sender, and set the subject of messages to influence their routing. A range of options are possible.

Suffice it to say that the subject of a message, whether set manually by you or automatically by a sender object, prescribes where the message will go.

Report a bug

2.8. Message Properties

Message properties are a list of key:value pairs that can be set for a message. Some predefined properties are used by the message broker to determine how to treat messages while they are in transit; these message properties can be set to ensure quality of service and guaranteed delivery. Other user-defined message properties can be set for application-specific functionality.

Report a bug

2.9. Connection

Connections in AMQP are network connections between the message broker and a message producer or message consumer.

Report a bug

2.10. Session

A *session* is a scoped conversation between a client application and the messaging broker. A session uses an connection for its communication, and it provides a scope for exclusive access to resources, and for the lifetime of a resource that is scoped to the session.

Note that multiple distinct sessions can use the same connection.

Report a bug

2.11. Exchange

In AMQP an *exchange* is a destination on the messaging broker that receives messages from senders. After receving a message, the exchange distributes a copy of the message to queues that are bound to the exchange. Consuming applications retrieve messages from those queues. Queues are bound to exchanges using binding keys that specify which messages from the exchange are of interest to the consumer. The queues buffer messages. This allows many consuming applications to receive messages from a single sender at different rates.

There are various types of exchanges that provide different distribution algorithms. The parameters used to bind queues to an exchange interact with the exchange's distribution algorithm to enable sophisticated routing schemas that are highly-performant.

Report a bug

2.12. Binding

Message queues are bound to exchanges using a *binding*. The binding is a description of which messages from the exchange are of interest to this queue. Different types of exchanges provide different distribution algorithms, so the content of the binding used to subscribe a queue to an exchange depends on the type of exchange as well as the interest of the subscriber.

Report a bug

2.13. Message Queue

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

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

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

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

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

Report a bug

2.14. Message Consumer (Receiver)

Message-consuming applications receive messages from the messaging broker. They do this by creating queues and binding them to an exchange on the messaging broker with a binding key.

Report a bug

Chapter 3. Getting Started

3.1. Getting Started with Python

3.1.1. Python Messaging Development

Python is a cross-platform dynamically interpreted language that is extremely easy to use for prototyping. Because it is interpreted and not compiled, the turn around time from coding to testing is fast. This makes it very good for testing and experimenting. It can be used like a scripting language, and can also be used for developing fairly large applications.

Many of the examples in this documentation use Python code to illustrate principles of programming messaging applications using Red Hat Enterprise Messaging. To run these sample programs is as simple as cutting and pasting the code into a file, then calling the `python` interpreter to execute the file.

Aside from the light-weight prototyping aspect, perhaps the most useful feature of Python for Messaging development is the ability to run the Python interpreter interactively. You can try things out and inspect the effect and state of objects in real-time.

The Python API for Apache Qpid is a first-class supported API in Red Hat Enterprise Messaging.

Report a bug

3.1.2. Python Client Libraries

There are three libraries for Python client development:

`python-qpid`
: Apache Qpid Python client library.

`python-qpid-qmf`
: Queue Management Framework (QMF) Python client library.

`python-saslwrapper`
: Python bindings for the saslwrapper library.

Report a bug

3.1.3. Install Python Client Libraries (Red Hat Enterprise Linux 5)

The Python client libraries for Red Hat Enterprise Linux 5 are available via the Red Hat Customer Portal.

If your machine uses *Red Hat Network classic* management you can install the Python client libraries via the `yum` command.

Subscribe your system to the `Red Hat MRG Messaging (for RHEL-5 Server) 2` channel.

Once your system is subscribed to this channel, with root privileges run the command:

```
yum install python-qpid python-qpid-qmf python-saslwrapper
```

Report a bug

3.1.4. Install Python Client Libraries (Red Hat Enterprise Linux 6)

The Python client libraries for Red Hat Enterprise Linux 6 are available via the Red Hat Customer Portal.

If your machine uses *Red Hat Network classic* management you can install the Python client libraries via the yum command.

The Python client libraries are in three base channels:

- Red Hat Enterprise Linux Server 6
- Red Hat Enterprise Linux Workstation 6
- Red Hat Enterprise Linux Client 6

Subscribe your system to one of the base channels.

When your system is subscribed to a base channel, with root privileges run the command:

```
yum install python-qpid python-qpid-qmf python-saslwrapper
```

Report a bug

3.1.5. Python "Hello World" Program Listing

Python

```
import sys
from qpid.messaging import *

connection = Connection("localhost:5672")

try:
  connection.open()
  session = connection.session()

  sender = session.sender("amq.topic")
  receiver = session.receiver("amq.topic")

  message = Message("Hello World!")
  sender.send(message)

  fetchedmessage = receiver.fetch(timeout=1)
  print fetchedmessage.content
  session.acknowledge()

except MessagingError,m:
    print m

connection.close()
```

See Also:

- Section 3.6.2, “"Hello World" Walk-through”

Report a bug

3.2. Getting Started with .NET

3.2.1. .NET Messaging Development

All .NET languages are supported using the C++ Messaging API. The most significant difference between .NET development and the other languages is that in a .NET environment the broker is always running on a remote server. With Python, C++, and Java development it is possible to run the broker and the client on the same machine during development, and the example code assumes this. All connections with .NET clients, however, are to a broker running remotely.

While developing and testing against a remote server it is important to configure the firewall correctly. This step can be skipped when the broker is running locally, but is crucial when the broker is running on a remote server.

Report a bug

3.2.2. Windows SDK

The MRG Messaging Windows SDK is a download containing necessary files for developing native C++ (unmanaged) and .NET (managed) clients for Windows.

Report a bug

3.2.3. Windows SDK Contents

The Windows SDK contains the following directories and files:

`\bin`

- Precompiled binary (.dll and .exe) files and the associated debug program database (.pdb) files
- Boost library files
- Microsoft Visual Studio 2008 MSVC90 runtime library files

`\include`

A directory tree of .h files

`\lib`

The linker .lib files that correspond to files in /bin

`\docs`

Apache Qpid C++ API Reference

`\examples`

A Visual Studio solution file and associated project files to demonstrate using the WinSDK in unmanaged C++

`\dotnet_examples`

A Visual Studio solution file and associated project files to demonstrate using the WinSDK in C#

`\management`
A python scripting code set for generating QMF data structures

Report a bug

3.2.4. Obtain the Windows SDK

Log in to the Red Hat Customer Portal.

Click on Downloads in the top menu, and select `Channels` from the sub-menu. The "Full Software Channel List" page appears.

Click the `Filter by Product Channel` combobox and select `Red Hat Enterprise MRG`, then click the `Filter` button. The Red Hat Enterprise MRG channels are returned, including `Red Hat MRG Messaging (for non-Linux) 2`.

Click the `IA-32` link next to the `Red Hat MRG Messaging (for non-Linux) 2` channel. The `MRG Messaging v. 2 (for non-Linux platforms)` channel page appears.

Click on the `Downloads` link underneath the Channel name (not the Downloads link the top menu).A list of available Windows SDK downloads appears.

Click on an available Windows SDK to download it.

Report a bug

3.2.5. Install Windows SDK on a 32-bit system

1. Obtain the 32-bit Windows SDK from the Red Hat Customer Portal.
2. Unzip the downloaded Windows SDK to your filesystem.
3. Run the Microsoft C++ Redistributable installer located in the `/bin` directory of the SDK.

Report a bug

3.2.6. Install Windows SDK on a 64-bit system

1. Obtain the 64-bit Windows SDK from the Red Hat Customer Portal.
2. Unzip the downloaded Windows SDK to your filesystem.
3. Download and install the 64-bit Microsoft C++ Redistributable installer from the Microsoft Download Center.

Report a bug

3.2.7. .NET C# "Hello World" Program Listing

The .NET binding for the Qpid C++ Messaging API applies to all .NET Framework managed code languages. C# is presented as an illustrative example.

C#/.NET

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

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();
                Session session = connection.CreateSession();

                Receiver receiver = session.CreateReceiver(address);
                Sender sender = session.CreateSender(address);

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

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

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

See Also:

- Section 3.6.2, ""Hello World" Walk-through"

Report a bug

3.3. Getting Started with C++

3.3.1. C++ Messaging Development

The open source Apache Qpid broker, on which Red Hat Enterprise Messaging is based, is available as a Java and as C++ broker. It is the C++ broker that is used to build Red Hat Enterprise Messaging.

There are some small differences between the Python and C++ APIs. Because the broker itself is written in C++, in those few areas where the C++ API differs from the Python API it is the general rule that the C++ API is the more fully-featured, and more extensively explored by users.

Report a bug

3.3.2. C++ on Linux

3.3.2.1. C++ Client Libraries

There are five packages for C++ client development:

`qpid-cpp-client`
: Apache Qpid C++ client library.

`qpid-cpp-client-ssl`
: SSL support for clients.

`qpid-cpp-client-rdma`
: RDMA Protocol support (including Infiniband) for Qpid clients.

`qpid-cpp-client-devel`
: Header files and tools for developing Qpid C++ clients.

`qpidd-cpp-client-devel-docs`
: AMQP client development documentation.

Report a bug

3.3.2.2. Install C++ Client Libraries (Red Hat Enterprise Linux 5)

The C++ client libraries for Red Hat Enterprise Linux 5 are available via the Red Hat Customer Portal.

If your machine uses *Red Hat Network classic* management you can install the C++ client libraries via the `yum` command.

Subscribe your system to the `Red Hat MRG Messaging (for RHEL-5 Server) 2` channel.

Once your system is subscribed to this channel, with root privileges run the command:

```
yum install qpid-cpp-client qpid-cpp-client-rdma qpid-cpp-client-ssl qpid-cpp-
client-devel
```

Report a bug

3.3.2.3. Install C++ Client Libraries (Red Hat Enterprise Linux 6)

The C++ client libraries for Red Hat Enterprise Linux 6 are available via the Red Hat Customer Portal.

If your machine uses *Red Hat Network classic* management you can install the C++ client libraries via the `yum` command.

Subscribe your system to the `Red Hat MRG Messaging v.2 (for RHEL-6 Server)` channel.

Once your system is subscribed to this channel, with root privileges run the command:

```
yum install qpid-cpp-client qpid-cpp-client-rdma qpid-cpp-client-ssl qpid-cpp-
client-devel
```

Report a bug

3.3.3. C++ on Windows

3.3.3.1. Windows SDK

The MRG Messaging Windows SDK is a download containing necessary files for developing native C++ (unmanaged) and .NET (managed) clients for Windows.

Report a bug

3.3.3.2. Windows SDK Contents

The Windows SDK contains the following directories and files:

`\bin`

- Precompiled binary (.dll and .exe) files and the associated debug program database (.pdb) files
- Boost library files
- Microsoft Visual Studio 2008 MSVC90 runtime library files

`\include`

A directory tree of .h files

`\lib`

The linker .lib files that correspond to files in /bin

`\docs`

Apache Qpid C++ API Reference

`\examples`

A Visual Studio solution file and associated project files to demonstrate using the WinSDK in unmanaged C++

`\dotnet_examples`

A Visual Studio solution file and associated project files to demonstrate using the WinSDK in C#

`\management`

A python scripting code set for generating QMF data structures

Report a bug

3.3.3.3. Obtain the Windows SDK

Log in to the Red Hat Customer Portal.

Click on Downloads in the top menu, and select `Channels` from the sub-menu. The "Full Software Channel List" page appears.

Click the `Filter by Product Channel` combobox and select `Red Hat Enterprise MRG`, then click the `Filter` button. The Red Hat Enterprise MRG channels are returned, including `Red Hat`

`MRG Messaging (for non-Linux) 2`.

Click the `IA-32` link next to the `Red Hat MRG Messaging (for non-Linux) 2` channel. The `MRG Messaging v. 2 (for non-Linux platforms)` channel page appears.

Click on the `Downloads` link underneath the Channel name (not the Downloads link the top menu).A list of available Windows SDK downloads appears.

Click on an available Windows SDK to download it.

Report a bug

3.3.3.4. Install Windows SDK on a 32-bit system

1. Obtain the 32-bit Windows SDK from the Red Hat Customer Portal.
2. Unzip the downloaded Windows SDK to your filesystem.
3. Run the Microsoft C++ Redistributable installer located in the `/bin` directory of the SDK.

Report a bug

3.3.3.5. Install Windows SDK on a 64-bit system

1. Obtain the 64-bit Windows SDK from the Red Hat Customer Portal.
2. Unzip the downloaded Windows SDK to your filesystem.
3. Download and install the 64-bit Microsoft C++ Redistributable installer from the Microsoft Download Center.

Report a bug

3.3.3.6. C++ "Hello World" Program Listing

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();
        Session session = connection.createSession();

        Receiver receiver = session.createReceiver(address);
        Sender sender = session.createSender(address);

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

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

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

See Also:

- Section 3.6.2, “"Hello World" Walk-through”

Report a bug

3.4. Getting Started with Java

3.4.1. Java Client Libraries

There are three libraries for Java client development:

`qpid-java-client`
: The Java implementation of the Qpid client

`qpid-java-common`
: Common files for the Qpid Java client

`qpid-java-example`
: Programming examples

Report a bug

3.4.2. Install Java Client Libraries (Red Hat Enterprise Linux 5)

The Java client development libraries for Red Hat Enterprise Linux 5 are available via the Red Hat Customer Portal.

1. Subscribe your system to the `Red Hat MRG Messaging v.2 (for RHEL 5 Server)` channel.
2. Install the Java client development libraries using the yum command:

   ```
   yum install qpid-java-client qpid-java-common qpid-java-example
   ```

Report a bug

3.4.3. Install Java Client Libraries (Red Hat Enterprise Linux 6)

The Java client development libraries for Red Hat Enterprise Linux 6 are available via the Red Hat Network.

To install the Java development packages:

1. Subscribe your system to the `Additional Services Channels for Red Hat Enterprise Linux 6 / MRG Messaging v.2 (for RHEL-6 Server)` channel.
2. Run the following yum command with root privileges:

   ```
   yum install qpid-java-client qpid-java-common qpid-java-example
   ```

Report a bug

3.4.4. Java JMS "Hello World" Program Listing

This program is available, along with other examples, in the `qpid-java-examples` package.

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"));
      Context context = new InitialContext(properties);

      ConnectionFactory connectionFactory
          = (ConnectionFactory) context.lookup("qpidConnectionfactory");
      Connection connection = connectionFactory.createConnection();
      connection.start();

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

      MessageProducer messageProducer = session.createProducer(destination);
      MessageConsumer messageConsumer = session.createConsumer(destination);

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

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

      connection.close();
      context.close();
    }
    catch (Exception exp) {
      exp.printStackTrace();
    }
  }
}
```

Here is the content of the Hello World example JNDI properties file, `hello.properties`:

```
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
```

Report a bug

3.5. Getting Started with Ruby

3.5.1. Ruby Messaging Development

The Ruby programming language does not have the same level of support as the other languages. There are libraries that allow you to access the Qpid Management Framework (QMF), but no supported client libraries for the standard messaging API.

Report a bug

3.5.2. Ruby Client Libraries

There are two libraries for Ruby client development:

`ruby-qpid-qmf`
: Ruby QMF bindings

`ruby-saslwrapper`
: Ruby bindings for the saslwrapper library

Note: There is a `ruby-qpid` package in Red Hat Enterprise Linux 6, but it is not compatible with the Qpid broker in MRG Messaging 2.

Report a bug

3.5.3. Install Ruby Client Libraries (Red Hat Enterprise Linux 5)

The Ruby client development libraries for Red Hat Enterprise Linux 5 are available via the Red Hat Customer Portal.

1. Subscribe your system to the `Red Hat MRG Messaging v.2 (for RHEL 5 Server)` channel.
2. Install the Ruby client development libraries using the yum command:

```
yum install ruby-qpid-qmf ruby-saslwrapper
```

Report a bug

3.5.4. Install Ruby Client Libraries (Red Hat Enterprise Linux 6)

The Ruby client development libraries are available via the Red Hat Customer Portal.

The `ruby-qpid-qmf` package is in the main channel; the `ruby-saslwrapper` package is in the Optional child channel.

To install the Ruby client development libraries:

1. Subscribe your system to one of the following channels:
   - `Red Hat Enterprise Linux Server 6`
   - `Red Hat Enterprise Linux Client 6`

   - `Red Hat Enterprise Linux Workstation 6`

2. With root privileges run the command:

   ```
   yum install ruby-qpid-qmf
   ```

3. Subscribe your system one of the following channels:
   - `Red Hat Enterprise Linux Optional Server v 6`
   - `Red Hat Enterprise Linux Optional Client 6`
   - `Red Hat Enterprise Linux Optional Workstation 6`

4. With root privileges run the command:

   ```
   yum install ruby-saslwrapper
   ```

Report a bug

3.6. Hello World

3.6.1. Red Hat Enterprise Messaging "Hello World"

Here is the "Hello World" example, showing how to send and receive a message with Red Hat Enterprise Messaging using the Qpid Messaging API.

Python

```
import sys
from qpid.messaging import *

connection = Connection("localhost:5672")

try:
  connection.open()
  session = connection.session()

  sender  = session.sender("amq.topic")
  receiver  = session.receiver("amq.topic")

  message = Message("Hello World!")
  sender.send(message)

  fetchedmessage = receiver.fetch(timeout=1)
  print fetchedmessage.content
  session.acknowledge()

except MessagingError,m:
    print m

connection.close()
```

C#/.NET

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

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();
                Session session = connection.CreateSession();

                Receiver receiver = session.CreateReceiver(address);
                Sender sender = session.CreateSender(address);

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

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

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

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();
        Session session = connection.createSession();

        Receiver receiver = session.createReceiver(address);
        Sender sender = session.createSender(address);

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

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

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

Report a bug

3.6.2. "Hello World" Walk-through

The Qpid Messaging client development libraries contain the functions we need to communicate with the messaging broker and create and manage messages, so our first task is to import them to our program:

Python

```
from qpid.messaging import *
```

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>

using namespace qpid::messaging;
```

C#/.NET

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

namespace Org.Apache.Qpid.Messaging {
```

To communicate with a message broker we need a connection. We get one by creating an instance of a `Connection` object. The Connection object constructor takes the url of the broker as its parameter:

Python

```
connection = Connection("localhost:5672")
```

C++

```
Connection connection(broker);
```

C#/.NET

```
Connection connection = null;
connection = new Connection(broker);
```

When you connect to a remote server that requires authentication you can provide a connection url in the form `username/password@serverurl:port`. If you try this with a remote server, remember to open the firewall on the message broker to allow incoming connections for the broker port.

Now that we have a Connection instance configured for our broker, the next step is to open the connection. The Connection object has an `open` method, which opens a configured connection.

Opening the connection might fail, if, for example, the message broker is off-line, so we will wrap it in a try:except block, and catch any exception.

Remember that Python uses indentation, so be careful with your spacing:

Python

```
try:
  connection.open()
```

C++

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

C#/.NET

```
connection.Open();
```

Now that we have an open connection to the server, we need to create a *session*. A session is a scoped conversation between our application and the server. The server uses the scope of the session to enforce exclusive access and session-scoped lifetimes of queues.

The `Connection` object has a `session` method that returns a `Session` object, so we get a session from the connection that we created previously:

Python

```
session = connection.session()
```

C++

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

C#/.NET

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

The `Session` object has `sender` and `receiver` methods, which take a target or source address as a parameter, and return a `Sender` and a `Receiver` object, respectively. These are the objects that we need to send and receive messages, so we will create them by calling the respective methods of our session. We will use the `amq.topic` exchange for this demo. This is a pre-configured exchange on the broker, so we don't need to create it, and we can rely on its presence:

Python

```
sender = session.sender("amq.topic")
  receiver = session.receiver("amq.topic")
```

C++

```
Receiver receiver = session.createReceiver(address);
  Sender sender = session.createSender(address);
```

C#/.NET

```
Receiver receiver = session.CreateReceiver(address);
Sender sender = session.CreateSender(address);
```

A sender can be thought of as a *router*. It routes messages from our application to the broker. The parameter we pass to the sender's constructor is the destination on the broker that our messages will be routed to. In this case, our sender will route messages to the `amq.topic` exchange on the broker. Because our routing target is an exchange, it will be routed further from there by the broker.

A receiver can be thought of as a *subscriber*. When we create a receiver, the parameter we pass to the constructor is resolved to an object on the server. If the object is a queue, then our receiver is subscribed to that queue. If the object is an exchange, as it is in this case, then a queue is created in the background and subscribed to the exchange for us. We will look in more detail at this later. For now, suffice it to say that our sender will send a message to the `amq.topic` exchange, and our receiver will receive it in a queue.

Now that we have a sender and a receiver, it's time to create a message to send. The `Message` object takes as a parameter to its constructor a string that becomes the `message.content`:

Python

```
message = Message("Hello World!")
```

The Message object constructor sets the correct content-type when you set the message.content through the constructor. However, if you set it after creating the Message object by assigning a value to the message.content property, then you also need to set the message.content_type property appropriately.

We can now use the send method of our sender to send the message to the broker:

Python

```
sender.send(message)
```

C++

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

C#/.NET

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

The message is sent to the amq.topic exchange on the message broker.

When we created our receiver, in the background the broker created a private temporary queue and subscribed it to the amq.topic exchange for us. The message is now waiting in that queue.

The next step is to retrieve the message from the broker using the fetch method of our receiver:

Python

```
fetchedmessage = receiver.fetch(timeout=1)
```

C++

```
Message message = receiver.fetch(Duration::SECOND * 1);
```

C#/.NET

```
Message message = new Message();
message = receiver.Fetch(DurationConstants.SECOND * 1);
```

The timeout parameter tells fetch how long to wait for a message. If we do not set a timeout the receiver will wait indefinitely for a message to appear on the queue. If we set the timeout to 0, the receiver will check the queue and return immediately if nothing is there. We set it to timeout in 1 second to ensure ample time for our message to be routed and appear in the queue.

We should now have a message, so we will print it out. Fetch returns a Message object, so we will print its content property:

Python

```
    print fetchedmessage.content
```

C++

```
std::cout << message.getContent() << std::endl;
```

C#/.NET

```
Console.WriteLine("{0}", message.GetContent());
```

To finish the transaction, acknowledge receipt of the message, which allows the message broker to clear it from the queue (dequeue the message):

Python

```
session.acknowledge()
```

C++

```
session.acknowledge();
```

C#/.NET

```
session.Acknowledge();
```

And finally, catch any exceptions and print something sensible to the console if they occur, and close our connection to the message broker:

Python

```
except MessagingError,m:
    print m

connection.close()
```

C++

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

C#/.NET

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

To run the program, save the file as `helloworld.py`, and then run it using the command `python` `helloworld.py`. If the message broker is running on your local machine, you should see the words: "Hello World!" printed on your programlisting.

Report a bug

Chapter 4. Beyond "Hello World"

4.1. Subscriptions

In the "Hello World" example, we sent a message to a *topic exchange*. AMQP messaging uses exchanges to provide flexible decoupled routing between message senders and message producers. Message consumers can *subscribe* to exchanges by creating a queue and binding it to the exchange. Exchanges and bindings are covered in more depth in their own sections. Here we will touch briefly on the topic exchange specifically, and learn something about the difference between exchanges and queues, as we learn how a message consumer *subscribes* to an exchange by binding a queue to it.

An exchange differs from a queue in a number of ways. One significant difference is that a queue will queue messages, and can store them, whereas an exchange will distribute them to queues, but has no local storage of its own. Message consumers are decoupled from the message producers by the message broker. Queues provide a mechanism for buffering messages between the two, to allow them to produce and consume data at different rates. A message consumer does not need to be connected at the point in time that a message is published to a queue to receive the message. The message remains in the queue until it is removed.

Exchanges, on the other hand, are a mechanism for routing messages to different queues. If a message is sent to an exchange and there are no queues bound to that exchange, then the message is lost, as there is no-one is listening and there is nowhere to store the message. Queues are subscriptions, and indicate to the broker that "*I (an application) am interested in these messages*", in the case of a queue created by a consumer, or "*I want these messages to be here for interested applications that are coming*", in the case of a queue created by a producer. To subscribe to messages of interest, an consumer application creates a queue and binds it to an exchange, or connects to an existing queue (*subscribe*). To provide messages that are of interest to applications, an application creates a queue and binds it to an exchange (*publish*). Consuming applications can then use that queue.

In our "Hello World" example program we created a receiver listening to the `amq.topic` exchange. In the background this creates a queue and `subscribes` it to the `amq.topic` exchange. Our Hello World program sender *publishes* to the `amq.topic` exchange. The `amq.topic` exchange is a good one to use for the demo. A topic exchange allows queues to be subscribed (to *bind* to the exchange) with a *binding key* that acts as a filter on the subject of messages sent to the exchange. Since we bind to the exchange with no binding key, we signal that we're interested in all messages coming through the exchange.

When our sender sends its message to the `amq.topic` exchange, the message is delivered to the subscription queue for our receiver. Our receiver then calls `fetch()` to retrieve the message from its subscription queue.

We will make two modifications to our Hello World program to demonstrate this.

First of all, we will send our message to the `amq.topic` exchange and *after* we send the message, register our receiver with the exchange.

We need to change the order of these operations:

Python

```
sender = session.sender("amq.topic")
receiver = session.receiver("amq.topic")

message = Message("Hello World!")
sender.send(message)
```

C++

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

Receiver receiver = session.createReceiver(address);
Sender sender = session.createSender(address);

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

C#/.NET

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

Receiver receiver = session.CreateReceiver(address);
Sender sender = session.CreateSender(address);

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

At the moment we register a receiver with the exchange *before* sending the message. Let's instead send the message, then register the receiver:

Python

```
sender = session.sender("amq.topic")

message = Message("Hello World!")
sender.send(message)

receiver = session.receiver("amq.topic")
```

C++

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

Sender sender = session.createSender(address);
sender.send(Message("Hello world!"));

Receiver receiver = session.createReceiver(address);
```

C#/.NET

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

Sender sender = session.CreateSender(address);
sender.Send(new Message("Hello world!"));

Receiver receiver = session.CreateReceiver(address);
```

When you run the modified Hello World program, you will not see the "Hello World!" message this time. What happened? The sender published the message to the `amq.topic` exchange.

The exchange then delivered the message to all the subscribed queues... which was none. When our receiver subscribes to the exchange it's too late to receive the message. In the original version of the program the receiver subscribes to the exchange before the message is sent, so it receives a copy of the message in its subscription queue.

Let's now examine the subscription queues that are created when we create the sender and receiver. We'll do that using the `qpid-config` command. Restart the broker to clear all the queues (all non-`durable` queues are destroyed when the broker restarts). Then run the command:

```
qpid-config queues
```

You see the list of queues on the broker.

Now modify the Hello World program back to its original form, where the receiver is created (*subscribed to the exchange*) before the message is sent. In order to see what happens, we'll pause the application between creating the exchange subscriptions and sending the message. We'll do that in Python by asking the user to press Enter, and using the `raw_input` method to grab some keyboard input.

Python

```
sender = session.sender("amq.topic")
receiver = session.receiver("amq.topic")

print "Press Enter to continue"
x= raw_input()

message = Message("Hello World!")
sender.send(message)
```

Now we run the program, and while it is paused, we use `qpid-config queues` to examine the queues on the broker.

Run the program, and while it is paused, issue the command:

```
qpid-config queues
```

You will see an exclusive queue with a unique random ID. This is the queue created and bound to the `amq.topic` exchange for us,to allow our receiver to receive messages from the exchange. You'll also see a number of other queues with the same ID number at the end of them. These are the queues that the `qpid-config` utility uses to query the message broker and receive the queue list you run the command. If you run the command again, you'll see that our receiver queue remains the same, and the other queues have a new ID - each time you run a `qpid-config` command it creates it own queues to receive a response from the server. You won't be able to see that those queues aren't there when you're not running `qpid-config`, because you need to run `qpid-config` to see the queues, but you can take my word for it.

Since the receiver's queue is bound to the exchange (*subscribed*) when the sender sends its message to the exchange, the "Hello World!" message is delivered to the subscription queue by the exchange, and is available for the receiver to fetch when it is ready.

The queue created for the receiver is an *exclusive queue*, which means that only one session can access it at a time.

Version 2.2 and below

To see the queue-exchange bindings, run:

```
qpid-config queues -b
```

The `-b` switch displays bindings. You'll see that the two dynamically created queues are bound to the `amq.topic` exchange.

Version 2.3 and above

To see the queue-exchange bindings, run:

```
qpid-config queues -r
```

The `-r` switch displays bindings. You'll see that the two dynamically created queues are bound to the `amq.topic` exchange.

When the application wakes up and completes execution, the call to `connection.close()` ends the session, and the two exclusive queues on the broker are deleted. You can run `qpid-config queues` again to verify that.

Another experiment you can try: create one receiver before the message is sent, and another receiver after the message is sent. We would expect the receiver created before the message is sent to receive the message, and the receiver created after the message is sent to not receive it.

Our simple application uses a dynamically created queue to interact with the `amq.topic` exchange. This queue is private (randomly named and `exclusive`), and deleted when the consumer disconnects, so it is not suitable for publishing. In order to make messages available to consumers who may or may not be connected to the exchange when the message is sent, a message-producing application needs to create a publicly-accessible queue (*publishing*). Consuming applications can then subscribe to this published queue and receive messages in a decoupled fashion.

Of course, if it's not important that your messages are buffered somewhere when no-one is listening, you can use the "Hello World" pattern of simply publishing to an exchange, and leave it to the consumers to create their own queues by subscribing to the exchange. AMQP messaging gives a lot of flexibility in messaging system design.

Report a bug

4.2. Publishing

As a message producer there are a number of different publishing strategies that you can use with AMQP messaging.

You can publish messages to an exchange, and message consuming applications can subscribe to the exchange, creating their own queues. There are a number of different exchange types that you can use, depending on how you want to distribute the information your application produces. One thing to note when publishing to an exchange is that if your message falls in the woods while no-one is listening, it doesn't make a sound: if no consumers are subscribed to the exchange when you send a message to it, the message disappears into the ether. It is not stored. If you need your messages to be stored whether consumers are listening or not, then you want to publish to a queue.

You can publish messages to a queue by creating a queue and subscribing it to an exchange. You then send messages to that exchange routed to that queue, and consuming applications can connect to your published queue and collect messages. This method of publishing is the one to use when your messages need to be stored on the broker whether someone is listening or not. Using this method of publishing, you can still allow consumers to create their own subscriptions to the exchange, or you can publish exclusively to your queue.

To publish to an exclusive queue, you would publish to a Direct Exchange, and bind your publishing queue to the exchange with an exclusive binding key. This means that you can route messages directly to your queue, and no-one else can bind a queue to the exchange that can receive those messages.

To publish to a queue and also allow consumers to create their own queues that receive your messages, you could publish to a Fanout or Topic exchange, and create and bind a queue with the appropriate binding key to receive your messages. Consumers can then subscribe to your queue, and can also create their own queues and bind them to the exchange.

Report a bug

4.3. AMQP Exchange Types

There are five AMQP Exchange types. The different exchanges provide different means of routing messages so that consumers can subscribe to the particular flow of information that is of interest to them.

The AMQP Exchange types are:

Direct

A Direct Exchange allows a consumer to bind a queue to it with a key. When a message is received by a direct-type exchange, the message is routed to any queues whose binding key matches the subject of the message. The Direct Exchange also supports exclusive bindings, which allow a queue to monopolise messages sent to an exchange, and implement a simple direct-to-queue model.

Topic

A Topic Exchange allows a consumer to bind a queue to it with a key that specifies wildcard matching. The wildcard is then matched against the subject of messages sent to the exchange. This allows you to implement message filtering patterns using a topic exchange and various queues with different binding keys.

Report a bug

4.4. Pre-configured Exchanges

Out of the box, the Red Hat Enterprise Messaging broker has five pre-configured exchanges that you can use for messaging. These exchanges are all configured as durable, so they are available whenever the broker is started:

Default exchange

A nameless direct exchange. All queues are bound to this exchange by default, allowing them to be accessed by queue name.

`amq.direct`

The pre-configured named direct exchange.

`amq.fanout`

The pre-configured fanout exchange.

`amq.match`
The pre-configured headers exchange.

`amq.topic`
The pre-configured topic exchange.

Report a bug

4.5. The Default Exchange

4.5.1. Default Exchange

The *Default Exchange* is a pre-configured nameless direct exchange.

All queues are bound to the Default Exchange by default. This means that a queue can be targeted by using the queue name as a target address, since a queue name unqualified with an exchange resolves to the nameless exchange.

Report a bug

4.5.2. Publish to a Queue using the Default Exchange

All queues automatically bind to the default exchange using the queue name as the binding key. So all you need to do to publish to a queue bound to the default exchange is to declare a queue. The binding to the Default Exchange is created automatically. Since the Default Exchange is a *direct exchange*, and is nameless, sending a message to the queue name is sufficient for it to arrive in your queue.

To create a queue named "quick-publish" bound to the Default Exchange using `qpid-config`:

```
qpid-config add queue quick-publish
```

In an application, queues can be created as a side-effect of creating a sender object. If the address contains the parameter `{create: always}` then the queue will be created if it does not already exist. In addition to `always`, the `create` command can also take the arguments `sender` and `receiver`, to indicate that the queue should be created only when a sender connects to the address, or only when a receiver connects to the address.

Here is the creation of the "quick-publish" example queue:

Python

```
sender = session.sender("quick-publish; {create: always}")
```

C++

```
Sender sender = session.createSender("quick-publish; {create: always}")
```

Report a bug

4.5.3. Subscribe to the Default Exchange

To subscribe to the Default Exchange, create a receiver and pass the name of the queue to the constructor. For example, to subscribe to the queue "quick-publish", using the Python API:

C++

```
Receiver receiver = session.createReceiver('quick-publish');
```

Python

```
receiver = session.receiver('quick-publish')
```

This receiver can now be used to retrieve messages from the `quick-publish` queue.

To obtain a browse-only view that does not remove messages from the queue:

C++

```
Receiver receiver = session.createReceiver('quick-publish; {mode: browse}');
```

Python

```
receiver = session.receiver('quick-publish; {mode: browse}')
```

If you want to create and subscribe a queue that does not yet exist, for example for your application to request its own copies of messages, use the `create` parameter:

C++

```
Receiver receiver = session.createReceiver("my-own-copies-please; {create:
always, node: {type: 'queue'}}");
```

Python

```
receiver = session.receiver("my-own-copies-please; {create: always, node:
{type: 'queue'}}")
```

If the queue "`my-own-copies-please`" already exists, then your receiver will connect to that queue. If the queue does not exist, then it will be created (all of this assumes sufficient privileges, of course).

One thing to bear in mind is that if an *exchange* named "`my-own-copies-please`" exists, your receiver will silently connect to that in preference to creating a queue. This is not what you intended, and will have unpredictable results. To avoid this, you can use the assert parameter, like this:

C++

```
try {
  Receiver receiver = session.createReceiver("my-own-copies-please; {create:
always, assert: always, node: {type: 'queue'}}");
} catch(const std::exception& error) {
  std::cerr << error.what() << std::endl;
}
```

Python

```
try:
  receiver = session.receiver("my-own-copies-please; {create: always, assert:
always, node: {type: 'queue'}}")
except MessagingError m:
  print m
```

Now if "`my-own-copies-please`" already exists and is an exchange, the receiver constructor will raise an exception: "`expected queue, got topic`".

Note that although it is an instance of a Direct Exchange, the Default Exchange does not allow multiple bindings using the same key. Each queue is bound to the Default Exchange uniquely. This means that you can only connect to a queue to get messages sent to it; you cannot bind another queue to the exchange in parallel to receive copies of the messages, as you can with other Direct Exchanges.

Report a bug

4.6. Direct Exchange

4.6.1. Direct Exchange

A Direct Exchange routes messages to queues where there is an exact match between the binding key of the queue and the subject of the message (*routing key*).

Note as you look at this picture that *multiple queues can bind to a Direct Exchange with the same binding key*. In the diagram we see one message going to one queue, but if other queues on that exchange have the same binding key, they will also receive the message.

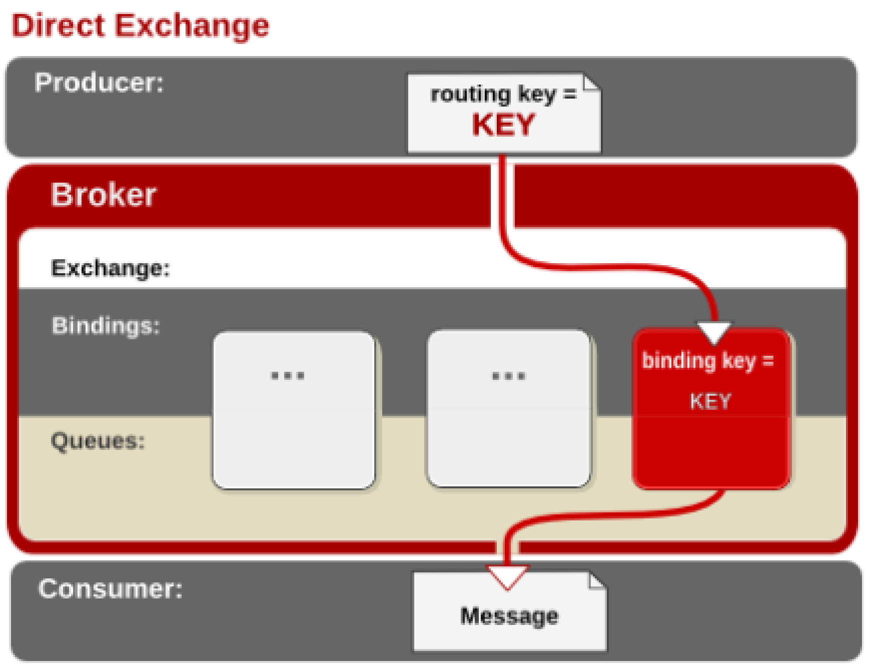

Figure 4.1. Direct Exchange

A Direct Exchange is a specialization of the Topic Exchange. Effectively, a Direct Exchange is a

Topic Exchange where there are no wildcards used, or allowed, on the binding key.

The Direct Exchange also supports *exclusive binding*, so that a queue can guarantee that it is the only recipient of messages sent to the Direct Exchange with the routing key used to exclusively bind the queue to the exchange.

Report a bug

4.6.2. Create a Direct Exchange using qpid-config

The command `qpid-config add exchange direct` *exchange name* creates a new direct exchange.

The following example `qpid-config` command creates a new direct exchange called `engineering`:

```
qpid-config add exchange direct engineering
```

Report a bug

4.6.3. Create a Direct Exchange from an application

You can creating a Direct Exchange in an application as a side effect of creating a sender or a receiver. For example, the following example creates a direct exchange called `engineering`:

Python

```
sender = session.sender('engineering;{create: always, node:{type:topic, x-
declare:{type:direct}}}')
```

In the case where an exchange named `engineering` already exists, the sender will not try to create a new one, but will connect to the existing one. You need to be careful, however, because if a *queue* with the name `engineering` already exists, then your sender will silently connect to that queue.

To ensure that your sender will connect to a new or existing exchange called `engineering`, you can use `assert`, as in this example:

Python

```
try:
  sender = session.sender('engineering;{create: always, node:{type:topic, x-
declare:{type:direct}}, assert: always}')
except MessagingError, m:
  print m
```

When you use `assert: always, node: {type: topic};` if `engineering` exists and is a queue, rather than an exchange, the sender constructor will raise an exception: "`expected topic, got` queue".

Note that while you can use `assert` to verify that it is an exchange and not a queue, you cannot verify what *type* of exchange it is.

Report a bug

4.6.4. Publish to a Direct Exchange

To publish to a direct exchange you have two options.

Create a sender that targets a specific endpoint

The first is to create a sender that routes messages directly to the endpoint that you wish to publish to. Remember that a Direct Exchange requires an exact match, so you are sending to a specific destination. At the same time, bear in mind that multiple queues can bind to the exchange to receive messages routed to the same destination. So it is a specific endpoint that may have multiple consumers.

This example creates a sender that will route messages to the `reports` endpoint on the `finance` exchange.

Python

```
sender = session.sender('finance/reports')
sender.send('Message to all consumers bound to finance with key reports')
```

Any messages now sent using `sender` will go to queues that have bound to the `finance` direct exchange using the key `reports`; with one caveat.

Let's look at our second option for publishing to a Direct Exchange, as it will help to explain this caveat.

Create a sender that targets the exchange

The second option is to create a sender that routes messages to the exchange, and use the message subject to control the routing to the specific endpoint. This way you can dynamically decide where messages will go, for example based on the names of keys that are provided at run-time, perhaps in the body of other messages.

This example demonstrates how this is done:

Python

```
sender = session.sender('finance; {assert: always, node: {type: topic}}')
msg = Message('Message to all consumers bound to finance with key reports')
msg.subject = 'reports'
sender.send(msg)
```

With a sender that targets the exchange, we specify where our message will go in the exchange by setting the `subject`. You can target different endpoints on that exchange by changing the subject before sending the message. For example, to send copies of the same message to `finance/reports` and `finance/records`:

Python

```
sender = session.sender('finance; {assert: always, node: {type: topic}}')
msg = Message('Message for reports and records')

msg.subject = 'reports'
sender.send(msg)

msg.subject = 'records'
sender.send(msg)
```

`{assert: always, node: {type: topic}}` is used to ensure that we don't inadvertently connect to a queue with the name `finance` bound to the default exchange. Queues and exchanges have separate namespaces, but remember that the default exchange is nameless.

A Caveat

As you can observe in the second case, setting the subject influences where the message is routed. If you use the first method — the sender with the subject in its address — you must be careful not to set the message subject inadvertently. The sender will write the correct subject into the message when you send it if the message subject is blank, but it will not overwrite any message subject that you provide. The first method — the sender with a subject in its address — provides a "default destination" for all messages it sends that do not have a message subject set. You can target other endpoints on the exchange by explicitly setting a subject before sending the message - in which case they go to the exchange for further routing based on your custom subject. Just be aware that setting the message subject determines its routing.

Report a bug

4.6.5. Subscribe to a Direct Exchange

There are three different patterns for subscribing to a Direct Exchange: a copy of messages, a shared queue, and an exclusive binding.

Copy of Messages

A copy of messages is where each consumer gets their own copy of every message. This arrangement makes sense, for example, when a service is logging activity based on messages, or when multiple consumers want notification of events.

Shared Queue

A shared queue is where multiple consumers connect to the same queue and take messages from the queue in a round-robin fashion. If consumer A and consumer B are accessing the same shared queue, then consumer A will not see the messages that consumer B takes from the queue. This arrangement makes sense, for example, in a scenario where worker nodes are dispatching jobs from a work queue. You want one node only to see each message.

These two patterns are not mutually exclusive - for example, three worker nodes could share a queue in round-robin fashion while another process gets its own copy of the messages in the queue to create an archive.

The two patterns can also be combined, for example: three worker nodes could share one queue, and two archiver nodes could share a second queue.

Exclusive Binding

The third pattern, exclusive binding, is where a consumer mandates that only consumer may have access to messages routed to an endpoint.

Subscribing to the Default Exchange using a Copy of Messages

This is the most straight-forward method to implement.

Subscribing to a Direct Exchange using a Shared Queue

To subscribe to a shared queue, you need to subscribe to a queue, rather than to the exchange. create a queue and bind it to the default exchange using a key. We can do that in one move using `x-bindings`. For example:

To subscribe to a shared queue, create a receiver and pass the name of the queue to the constructor. For example, to obtain access to the queue "reports" on the "finance" direct exchange:

Python

```
receiver = session.receiver('finance/reports')
```

This receiver can now be used to retrieve messages from the `quick-publish` queue.

To obtain a browse-only view that does not remove messages from the queue:

Python

```
receiver = session.receiver('quick-publish; {mode: browse}')
```

Python

```
receiver = session.receiver("my-own-copy; {create: always, node:{type:queue,
x-bindings: [{exchange: '', queue: 'my-own-copy', key: 'quick-publish'}]}}")
```

We have created a queue named "`my-own-copy`" and bound it to the default exchange with the key "`quick-publish`".

Report a bug

4.6.6. Exclusive Bindings for Direct Exchanges

Declaring an *exclusive binding* on a direct exchange ensures that a maximum of one consumer is bound to the exchange using this key at any time. When a new consumer is subscribed to the exchange using this key, the previous consumer's binding is dropped synchronously. This allows messaging routing to be switched between consumers with guaranteed message atomicity, with no possibility of dropped messages or duplicate delivery while the composite bind/unbind operation is taking place.

The exchange-bind argument `qpid.exclusive-binding` is used to declare an exclusive binding.

C++

```
    FieldTable args;
    args.setInt("qpid.exclusive-binding",1);

    //the following will cause the only binding from amq.direct with 'my-key'
    //to be the one to 'my-queue'; if there were any previous bindings for that
    //key they will be removed. This is atomic w.r.t message routing through
the
    //exchange.
    session.exchangeBind(arg::exchange="amq.direct", arg::queue="my-queue",
                         arg::bindingKey="my-key", arg::arguments=args);
```

Report a bug

4.7. Fanout Exchange

4.7.1. The pre-configured Fanout Exchange

Red Hat Enterprise Messaging ships with a pre-configured Fanout exchange named `amq.fanout`.

Report a bug

4.7.2. Fanout Exchange

A Fanout Exchange routes all messages to all queues bound to the exchange.

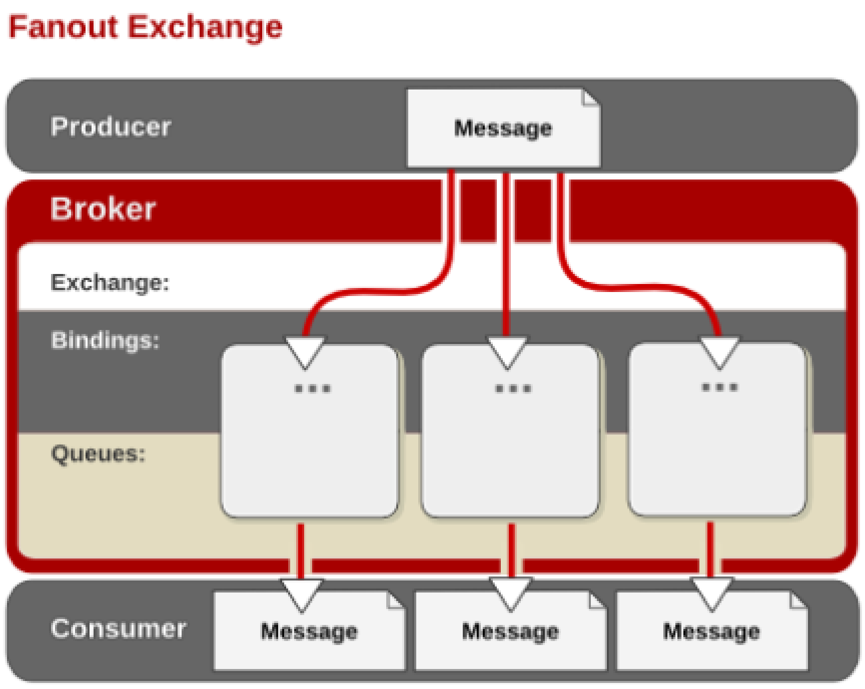

Figure 4.2. Fanout Exchange

A Fanout exchange is a specialization of the Topic Exchange. Effectively, a Fanout Exchange is a Topic Exchange where all queues bound to the exchange use a wildcard of `*` as their binding key.

Report a bug

4.7.3. Create a Fanout Exchange using qpid-config

The following example creates a new fanout exchange using `qpid-config`:

```
qpid-config add exchange fanout my-fanout-exchange
```

To make the exchange `durable` (persistent between restarts of the broker), use the `--durable` option:

```
qpid-config add exchange fanout my-fanout-exchange --durable
```

The `qpid-config exchanges` command lists the exchanges on the broker.

Report a bug

4.7.4. Create a Fanout Exchange from an application

A fanout exchange can be declared in an application by using the following parameters in the address of a sender or receiver:

- `create: always`
- `node: {type: topic, x-declare: {exchange: `*`exchange-name`*`, type: fanout}}`

The following example presents the address to create a new fanout exchange named `myfanout`.

Python

```
tx = ssn.sender("myfanout; {create: always, node: {type: topic, x-declare:
{exchange: myfanout, type: fanout}}}")
```

Report a bug

4.7.5. Publish to Multiple Queues using the Fanout Exchange

All queues bound to a fanout exchange receive a copy of all messages sent to the exchange; so to publish to all consumers on a fanout exchange, send a message to the exchange.

Python

```
import sys
from qpid.messaging import *
con = Connection("localhost:5672")
con.open()
try:
  ssn = con.session()
  tx = ssn.sender("amq.fanout")
  tx.send("Hello to all consumers bound to the amq.fanout exchange")
finally:
  con.close()
```

Report a bug

4.7.6. Subscribe to a Fanout Exchange

When subscribing to a fanout exchange you have two options:

1. Subscribe to the exchange using an ephemeral subscription. This creates and binds a temporary private queue that is destroyed when your application disconnects. This approach makes sense when you do not need to share responsibility for the messages between multiple consumers, and you do not care about messages that are sent when your application is not running or is disconnected.
2. Subscribe to a queue that is bound to the exchange. This allows messages to be buffered in the queue when your application is disconnected, and and allows several consumers to share responsibility for the messages in the queue.

Private, ephemeral subscription

To implement the private, ephemeral subscription, simply create a receiver using the name of the fanout exchange as the receiver's address. For example:

Python

```
rx = receiver("amq.fanout")
```

Shareable subscription

To implement a shareable subscription that persists across consumer application restarts, create a queue, and subscribe to that queue.

You can create and bind the queue using `qpid-config`:

```
qpid-config add queue shared-q
qpid-config bind amq.fanout shared-q
```

Note: To make the queue persistent across broker restarts, use the `--durable` option.

Use the `qpid-config` command to view the exchange bindings after issuing these commands. On MRG Messaging 2.2 and below use the command `qpid-config exchanges -b`. On MRG Messaging 2.3 and above use the command `qpid-config exchanges -r`.

Once you have created and bound the queue, in your application create a receiver that listens to this queue:

Python

```
rx = receiver("shared-q")
```

You could also create and bind the queue in the application code, rather than using `qpid-config`:

Python

```
rx = receiver("shared-q;{create: always, link: {x-bindings: [{exchange:
'amq.fanout', queue: 'shared-q'}]}}")
```

Report a bug

4.8. Topic Exchange

4.8.1. The pre-configured Topic Exchange

Red Hat Enterprise Messaging ships with a pre-configured durable topic exchange named `amq.topic`.

Report a bug

4.8.2. Topic Exchange

A Topic Exchange routes messages based on the routing key (subject) of the message and the binding key of the subscription, just as a direct exchange does. The difference is that a topic exchange supports the use of wildcards in binding keys, allowing you to implement flexible routing schemas.

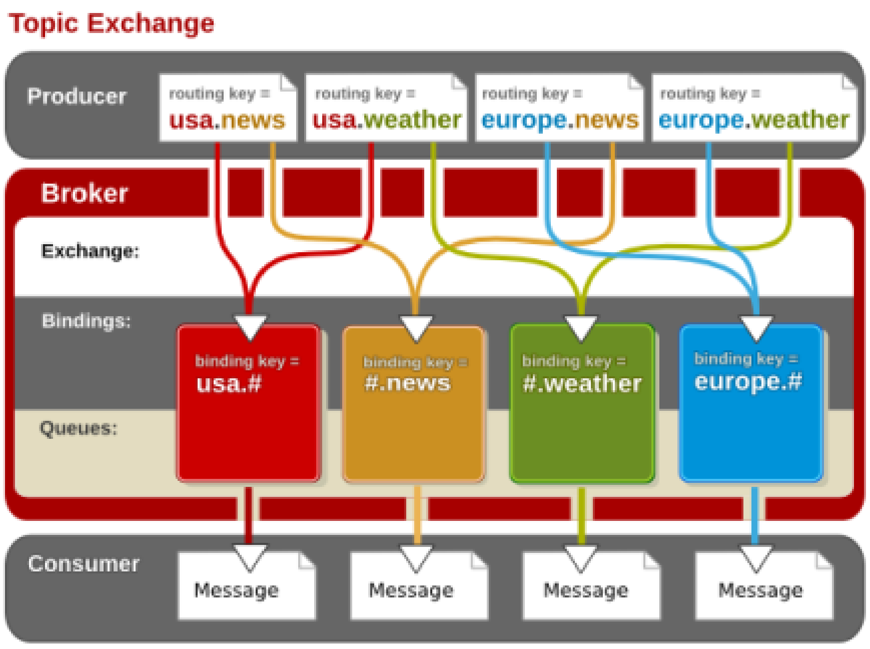

Figure 4.3. Topic Exchange

Wildcard matching and Topic Exchanges

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

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

Report a bug

4.8.3. Create a Topic Exchange using qpid-config

The following qpid-config command creates a topic exchange called news:

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

Report a bug

4.8.4. Create a Topic Exchange from an application

The following example creates a topic exchange called news:

Python

```
txtopic = ssn.sender("news; {create: always, node: {type: topic}}")
```

Report a bug

4.8.5. Publish to a Topic Exchange

To publish to a topic exchange, create a sender whose address is the exchange, then set the subject of the message to the routing key.

In the following example, messages are sent to the news topic exchange with routing keys that allow geography-based subscriptions by consumers:

Python

```
import sys
from qpid.messaging import *
conn = Connection("localhost:5672")
conn.open()
try:
  ssn = conn.session()
  txnews = ssn.sender("news; {create: always, node: {type: topic}}")
  msg = Message("News about Europe")
  msg.subject = "europe.news"
  txnews.send(msg)
  msg = Message("News about the US")
  msg.subject = "usa.news"
  txnews.send(msg)
finally:
  conn.close()
```

Report a bug

4.8.6. Subscribe to a Topic Exchange

To subscribe to topic exchange, create a queue and bind it to the exchange with the desired routing key.

The following example uses `qpid-config` to create a queue named news and bind it to the `amq.topic` exchange with a wildcard that matches *everything*.news, where *everything* is any number of period-separated terms:

```
qpid-config add queue news
qpid-config bind amq.topic news "#.news"
```

Now you can listen to the news queue for all messages whose routing key ends with .news:

Python

```
rxnews = ssn.receiver("news")
```

You can also do the entire operation (create, bind, and listen) in code, by using an address like the one in the following example:

Python

```
rxnews = ssn.receiver("news;{create: always, node: {type:queue}, link:{x-
bindings:[{exchange: 'amq.topic', queue: 'news', key: '#.news'}]}}")
```

You could also create an ephemeral subscription for your application, if you do not care about queuing messages when your application is disconnected or sharing responsibility for messages. This method creates and binds a temporary private queue for your application:

Python

```
rxnews = ssn.receiver("amq.topic/#.news");
```

In topic exchange binding key wildcard matching, the # symbol will match any number of period-separated terms. The # will match exactly one term.

Report a bug

4.9. Headers Exchange

4.9.1. The pre-configured Headers Exchange

Red Hat Enterprise Messaging ships with a pre-configured durable headers exchange named `amq.match`.

Report a bug

4.9.2. Headers Exchange

The Headers Exchange allows routing based on matches with properties in the message header. This allows flexible routing schemas based on arbitrary domain-specific attributes of messages.

Report a bug

4.9.3. Create a Headers Exchange using qpid-config

The following example `qpid-config` command creates a headers exchange called `property-match`:

```
qpid-config add exchange headers property-match
```

Report a bug

4.9.4. Create a Headers Exchange from an application

The following code creates a headers exchange called `headers-match`:

Python

```
txheaders = ssn.sender("headers-match;{create: always, node: {type: topic, x-
declare: {exchange: headers-match, type: headers}}}")
```

Report a bug

4.9.5. Publish to a Headers Exchange

To publish to a headers exchange, pass the name of the exchange to the sender constructor, and add the header keys and value to the message `properties`. For example:

Python

```
import sys
from qpid.messaging import *
conn =  Connection("localhost:5672")
conn.open()
try:
  ssn = conn.session()
  txheaders = sender("amq.match")
  msg = Message("Headers Exchange message")
  msg.properties['header1'] = 'value1'
  txheaders.send(msg)
finally:
  ssn.close()
```

Report a bug

4.9.6. Subscribe to a Headers Exchange

The following code creates a queue `match-q`, and subscribes it to the `amq.match` exchange using a binding key that matches messages that have a header key `header1` with a value of either `value1` or `value2`:

Python

```
rxheaders = ssn.receiver("match-q;{create: always, node: {type: queue},
link:{x-bindings:[{key: 'binding-name', exchange: 'amq.match', queue: 'match-q',
arguments:{'x-match': 'any', 'header1': 'value1', 'header1' : 'value2'}}]}}")
```

The `x-match` argument can take the values `any`, which matches messages with *any* of the key value pairs in the binding, or `all`, which matches messages that have *all* the key value pairs from the binding key in their header.

Note the `x-bindings` argument `key`. This argument creates a named handle for the binding, which is visible when running `qpid-config exchanges -b` on MRG Messaging version 2.2 or earlier, or the command `qpid-config exchanges -r` on MRG Messaging version 2.3 or later. Without a handle, a binding cannot be deleted by name. A `null` key is valid, but in addition to not being able to be deleted by name, when a binding is created with a `null` handle, any further attempt to create a binding with a `null` handle on that exchange will be update the existing binding rather than create a new one.

Report a bug

4.10. XML Exchange

4.10.1. Custom Exchange Types

AMQP Messaging supports custom exchange types. Custom exchanges allow you to manipulate or match messages based on any criteria.

Red Hat Enterprise Messaging ships with one custom exchange type, the *XML Exchange*.

Report a bug

4.10.2. The pre-configured XML Exchange Type

Red Hat Enterprise Messaging ships with a custom XML Exchange *type*.

The XML Exchange matches messages based on a XQuery applied to the headers or message content. Messages containing XML data can be sent to this exchange and filtered based on the message contents, as well as on the message headers.

Report a bug

4.10.3. Create an XML Exchange

The following example `qpid-config` command creates an XML exchange called `myxml`:

```
qpid-config add exchange xml myxml
```

The following example code demonstrates how to achieve the same in an application:

Python

```
tx = ssn.sender("myxml; {create: always, node: {type: topic, x-declare:
{exchange: myxml, type: xml}}}")
```

Report a bug

4.10.4. Subscribe to the XML Exchange

The following code subscribes to an XML exchange `myxml` by creating a queue `xmlq` and binding it to the exchange with an XQuery.

Python

```
rxXML = ssn.receiver("myxmlq; {create:always, link: { x-bindings:
[{exchange:myxml, key:weather, arguments:{xquery:'./weather'} }]}}")
```

The XQuery `./weather` will match any messages whose body content has the root XML element `<weather>`.

Note the use of the `key` argument for `x-bindings`. This ensures that the binding has a unique name, allowing it to be deleted and updated by name, and ensuring that it is not accidentally updated, as might be the case if it were anonymous in the namespace of the exchange.

The following code demonstrates using the XML exchange with a more complex XQuery:

Python

```
#!/usr/bin/python
import sys
from qpid.messaging import *

conn = Connection("localhost:5672")
conn.open()
try:
  ssn = conn.session()
                  tx = ssn.sender("myxml/weather; {create: always, node:
{type: topic, x-declare: {exchange: myxml, type: xml}}}")

  xquerystr = 'let $w := ./weather '
  xquerystr += "return $w/station = 'Raleigh-Durham International Airport
(KRDU)' "
  xquerystr += 'and $w/temperature_f > 50 '
  xquerystr += 'and $w/temperature_f - $w/dewpoint > 5 '
  xquerystr += 'and $w/wind_speed_mph > 7 '
  xquerystr += 'and $w/wind_speed_mph < 20'

  rxaddr = 'myxmlq; {create: always, '
  rxaddr += 'link: {x-bindings: [{exchange: myxml, '
  rxaddr += 'key: weather, '
  rxaddr += 'arguments: {xquery: "' + xquerystr + '"'
  rxaddr += '}}]}}'

  rx = ssn.receiver(rxaddr)
  rx = ssn.receiver('myxmlq; {create: always, link: {x-bindings: [{exchange:
myxml, key: weather, arguments: {xquery: "./weather"}}]}}')
  print rxaddr

  msgstr = '<weather>'
  msgstr += '<station>Raleigh-Durham International Airport (KRDU)</station>'
  msgstr += '<wind_speed_mph>16</wind_speed_mph>'
  msgstr += '<temperature_f>70</temperature_f>'
  msgstr += '<dewpoint>35</dewpoint>'
  msgstr += '</weather>'

  msg = Message(msgstr)

  tx.send(msg)

  rxmsg = rx.fetch(timeout=1)
  print rxmsg

  ssn.acknowledge()

finally:
  conn.close()
```

Report a bug

Chapter 5. Message Delivery and Acceptance

5.1. The Lifecycle of a Message

5.1.1. Message Delivery Overview

The following diagram illustrates the message delivery lifecycle.

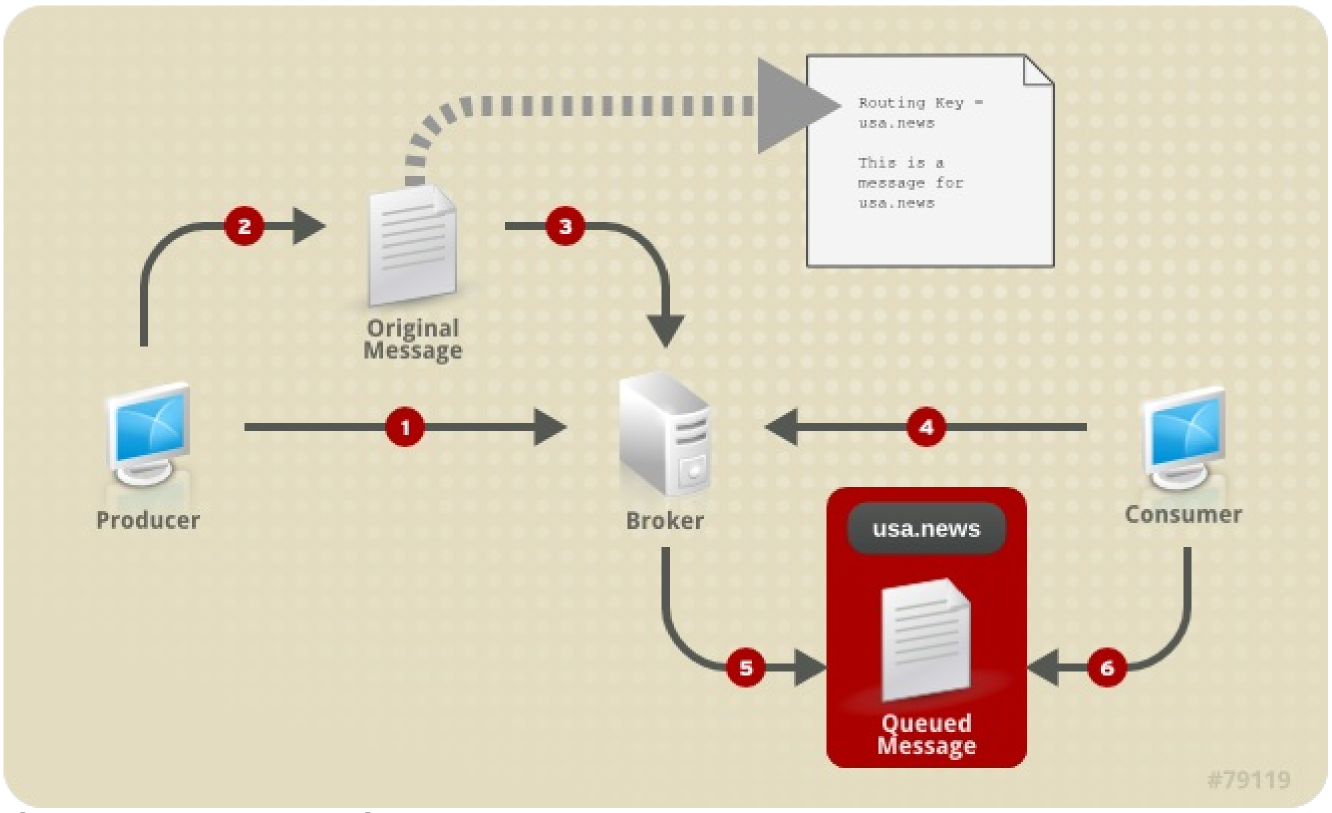

Figure 5.1. Fanout Exchange

A message producer generates a message. A message is an object with content, a subject, and headers. At the minimum, a message producer will produce a message with message content.

The message producer may send the message to the broker and let the routing be taken care of by the properties of the message or by the address of the sender object used to send the message (1).

Or the message producer may set the `message.subject`, which acts as the routing key (2), and then send the message to the broker (3).

Consumers subscribed to exchanges (which uses a temporary, private queue in the background) receive messages when they are connected (4).

Messages are buffered in queues that are subscribed to exchanges (5). Consumers can subscribe to queues and receive messages that were buffered while the consumer was disconnected (6). These queues can also be used to share responsibility for messages between consumers.

Report a bug

5.1.2. Message Generation

The `Message` object is used to generate a message.

Python

```
import sys
from qpid.messaging import *
...
msg = Message('This is the message content')
msg.content = 'Message content can be assigned like this'
msg.properties['header-key'] = 'value'

tx = ssn.sender('amq.topic')

# msg.subject set by sender for routing purposes
tx.send(msg)

msg.subject = 'Messaging Routing Key can also be manually set'
# beware that this will interfere with sender-object-based routing
```

Report a bug

5.1.3. Message Send over Reliable Link

When sent over a reliable link:

1. The sender passes the message to the broker.
2. The broker responds with an acknowledgement that it takes responsibility for delivery of the message.
3. The sender deletes its local copy of the message.

In synchronous operation the thread is blocked while this acknowledgement round-trip occurs. When sending using asynchronous operation, the acknowledgement and deletion is performed in the background, and sent but unacknowledged messages are buffered in the sender replay buffer until they are acknowledged.

Report a bug

5.1.4. Message Send over Unreliable Link

When sent over an unreliable link:

1. The sender passes the message to the broker.
2. The sender deletes the local copy of the message.

Messages may be lost between the sender and the broker in this mode.

Report a bug

5.1.5. Message Distribution on the Broker

When the broker receives a message, it examines the message and the routing information associated with it to determine how to deliver it.

The bindings on the exchanges that receive the message are examined, and when there is a match between the message and a binding, the message is delivered to any queue with that binding.

Report a bug

5.1.6. Message Receive over Reliable Link

When a message is received over a reliable link:

1. The broker passes the message to the receiver.

From this point a number of possibilities exist when the receiver is an acquiring consumer:

1. The receiver *acknowledges* responsibility for the message. In this case the broker deletes the server-side copy of the message.
2. The receiver *rejects* the message. In this case the broker routes the message to an `alternate-exchange` if one is defined for the queue, or else discards the message.
3. The receiver *releases* the message. In this case the broker returns the message to the queue with a message header `redelivered:true`.
4. The receiver disconnects without acknowledging or rejecting the message. In this case the broker returns the message to the queue with a message header `redelivered:true`.

Report a bug

5.1.7. Message Receive over Unreliable Link

When a message is received over an unreliable link:

1. The broker passes the message to the receiver.
2. The broker deletes the server-side copy of the message.

There is no opportunity for the receiver to reject the message, and no opportunity for the broker to redeliver it when using an unreliable link.

Report a bug

5.2. Browsing and Consuming Messages

5.2.1. Message Acquisition and Acceptance

A message consumer can *browse* the messages in a queue, or *consume* them.

Browsing means that the consuming application reads the messages, but the messages remain on the queue for other consumers. *Consuming* means that the consuming application removes the message from the queue. This is also known as *acquiring* a message.

We will first look at the broad distinction between browsing and acquiring messages, then in Acquired and Acknowledged we'll look in more detail at the acquisition process, which has two phases that we need to understand.

Browsing

The included `drain` program can be used in either browse or acquisition mode.

The drain source code is part of the C++ and the Python client library packages. You can compile the C++ source code, or run the Python source uncompiled using a Python interpreter.

When the client library packages are installed, `drain` can be found in:

```
/usr/share/doc/python-qpid-0.14/examples/api/drain
/usr/share/qpidc/examples/messaging/drain.cpp
```

To demonstrate the difference between browsing and acquisition, you can try the following:

With the broker installed and running, create a queue with the `qpid-config` command:

```
qpid-config add queue browse-acquire-demo
```

You should now see your `browse-acquire-demo` queue when you run `qpid-config queues`.

Now let's send a message to the `browse-acquire-demo` using `spout`. `Spout` is included in the same packages as `drain`, and can be found in the same directories. Run spout to send a message to the queue:

```
./spout browse-acquire-demo "Hello World"
```

Our "Hello World" message has now been sent to the `browse-acquire-demo` queue. Let's use drain to *browse* it first of all:

```
./drain -c 0 "browse-acquire-demo; {mode:browse}"
```

You will now see the "Hello World" message. Run the above `drain` a second time, and you'll see the message again. Running the drain program twice simulates two different browsing consumers accessing the queue. The message is read and remains available for other consuming applications when it is browsed.

Try deleting the `browse-acquire-demo` queue using `qpid-config`:

```
qpid-config del queue browse-acquire-demo
```

`qpid-config` responds with an error because a message remains in the queue.

Now run this drain command:

```
./drain -c 0 "browse-acquire-demo"
```

The default mode is *acquisition*. When drain is run like this with no mode specified, it *acquires* the message. You will see the "Hello World" message just as you did on the previous browsing accesses. However, this time the message has been removed. Try browsing it again using drain. The queue is empty.

You can delete the now-empty queue using `qpid-config`:

```
qpid-config del queue browse-acquire-demo
```

One thing you will not see with the `drain` demo is the fact that browsers see a message only once. Because each time `drain` is run it creates a different browser, it sees the message in the queue each time. The same browser, however, sees the message only once, no matter how many times it looks.

The following Python code demonstrates browsing and acquiring, and demonstrates how a browser sees each message once:

Python

```
import sys
from qpid.messaging import *

def msgfetch(rx):
  try:
    msg = rx.fetch(timeout=1)
  except MessagingError, m:
    msg = m
  return msg

connection = Connection("localhost:5672")
connection.open()
try:
  session = connection.session()
  tx = session.sender("browse-acquire-demo;{create:always}")
  rxbrowse1 = session.receiver("browse-acquire-demo;{mode:browse}")
  rxbrowse2 = session.receiver("browse-acquire-demo;{mode:browse}")
  rxbrowse3 = session.receiver("browse-acquire-demo;{mode:browse}")
  rxacquire = session.receiver("browse-acquire-demo")

  tx.send("Hello World")

  print "\nBrowser 1 saw message:"
  print msgfetch(rxbrowse1)

  print "Browser 1 then saw message:"
  print msgfetch(rxbrowse1)

  print "\nBrowser 2 saw message:"
  print msgfetch(rxbrowse2)

  print "Browser 2 then saw message:"
  print msgfetch(rxbrowse2)

  print "\nAcquired message:"
  print msgfetch(rxacquire)

  print "\nBrowser 3 saw message:"
  print msgfetch(rxbrowse3)

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

Browser 1 and Browser 2 both see the message, and only see it once each. Because the message is acquired before Browser 3 looks at the queue, Browser 3 sees no message on the queue.

However, now run `drain` to examine the queue:

```
./drain -c 0 browse-acquire-demo
```

You may be surprised to see the message still on the queue (you just removed it, by the way). What happened?

Acquired and Acknowledged

When our receiver acquired the message from the queue, the broker set the message to `acquired`. When a message is `acquired`, the broker treats the message as if it has been delivered, but it does not delete it from the queue. One of a number of things happen from here: the consumer who acquired the message *acknowledges* the message, *releases* the message, or *rejects* the message, or the consumer might disconnect through a network failure.

In our case, our application is disconnecting from the broker without acknowledging receipt of the message. While our application is connected the message is `acquired`, and message consumers browsing or fetching from the queue will not see the message. When our application disconnects without acknowledging receipt, the broker switches the message out of `acquired` state and sets a header `redelivered=True`. The message is then made available to other consumers, such as the `drain` browser that we ran after our application closed.

This goal of the "acquire, acknowledge" pattern is to provide reliable delivery of messages. Imagine a situation where a group of nodes are performing a service that is driven by messages. Each node in the workgroup grabs a bunch of messages from the queue when it has the capacity to perform some work. A node might grab a handful of messages from the queue, and then suffer a power outage. In this case those messages would be missing, if the broker did not have the concept of acquire and acknowledge. With this pattern, the worker node can acquire the messages, perform some work, and then acknowledge ownership at a point in time where it is safe to say that the message has been delivered and acted on. This narrows the window for exceptions. Even in the case where the node fails right at the critical moment after it has acted on the messages but before it can acknowledge receipt, the other nodes will retrieve the messages from the queue with the header '`redelivered=true`'. This alerts the other nodes that this message *may* have already been acted on, and they can perform checks to see if that is so. This narrows the window for exceptions even further, when the applications are designed to take advantage of these features.

To see a message returning to the queue when a consumer disconnects without acquiring the message demonstrated inside the application, add the following code to the end of the application, after the final `connection.close()` line:

Python

```
connection.open()
try:
  session=connection.session()

  rxacquire2 = session.receiver("browse-acquire-demo")
  print "\nAcquirer 2 saw message:"
  print msgfetch(rxacquire2)
except MessagingError, m:
  print m
finally:
  session.acknowledge()
  connection.close()
```

Our application closes its connection, disconnecting the consumer from the broker without acknowledging receipt of the message. We then open a new connection to broker, effectively appearing as a new consumer. Our receiver now sees the message, which has been marked by the broker as `redelivered` to inform us that another consumer acquired this message previously. We have now acquired this message, and it will again disappear for other consumers browsing or fetching from this queue. This time, however, we call `session.acknowledge()` before closing the connection. This method acknowledges receipt of the message (it acknowledges all messages as-yet unacknowledged for the session). Since we have acknowledged receipt of the message, the message is `acquired`, and it is removed from the queue.

If you run `drain` now, you will see that there are no messages in the queue.

Releasing a message

A consumer can explicitly *release* a message. When this happens, the message is returned to the queue for redelivery. The effect is the same as if the consumer lost its connection to the broker.

To release the message explicitly with the Python API, call the `acknowledge()` method with the message and `Disposition(RELEASED)` as parameters:

```
session.acknowledge(msg, Disposition(RELEASED))
```

To release the message explicitly with the C++ API, call the session's `release()` method.

Link Reliability

Note that this two-phase acquisition and acceptance behavior is the behavior over a *reliable* link (technically an *at-least-once* link), which is the default link for receiver connections to the broker. If you explicitly connect your receiver to a queue using an *unreliable* link, or directly connect to an exchange, then received messages are immediately acquired with no need to acknowledge them.

Cleaning up the demo queue

To delete the queue we used for this demo, you can either restart the broker (all non-`durable` queues are deleted when the broker is restarted), or you can use `qpid-config`:

```
qpid-config del queue browse-acquire-demo
```

If there are messages remaining in the queue this command will fail with an message informing you that the queue is not empty. You can use the `--force` switch to override this check and delete a queue with messages in it, or you can use `drain` to empty the queue, and then reissue the command on the now-empty queue.

Report a bug

5.2.2. Message Acquisition and Acceptance on an Unreliable Link

The default link between a receiver and the broker is a reliable link (technically known as a link with *at-least-once* reliability). This link uses a two-phase acquire and acknowlege behavior to ensure that the responsibility for a message is explicitly accepted by a consumer before the broker deletes it from the queue.

You can also request an *unreliable* link between the receiver and the broker. Over an unreliable link, messages are considered acknowledged and acquired as soon as the consumer fetches them from the queue. There is no acquired phase where a message will return to the queue if the receiver does not explicitly acknowledge it. The broker considers that the consumer has acknowledged the acquisition and deletes the message when the consumer fetches it, without waiting for an acquisition acknowledgement. This link has reduced reliability, but can result in increased throughput. It is useful when you can afford to lose messages in the event of consumer failure.

To request an unreliable link, specify `link: {reliability: unreliable}` in the address. For example, to create a receiver with an unreliable link to a queue named "browse-acquire-demo":

Python

```
rxacquire = session.receiver("browse-acquire-demo; {link:{reliability:
unreliable}")
```

The following program demonstrates the use and behavior of receivers using an unreliable link:

Python

```
import sys
from qpid.messaging import *

def msgfetch(rx):
  try:
    msg = rx.fetch(timeout=1)
  except MessagingError, m:
    msg = m
  return msg

linktype=""
while linktype != "R" and linktype !="U":
  response = raw_input("Use (R)eliable or (U)nreliable link [R/U]?")
  linktype = response.upper()

connection = Connection("localhost:5672")
connection.open()
try:
  session = connection.session()
  tx = session.sender("browse-acquire-demo;{create: always}")
  rxbrowse1 = session.receiver("browse-acquire-demo;{mode:browse}")
  rxbrowse2 = session.receiver("browse-acquire-demo;{mode:browse}")
  rxbrowse3 = session.receiver("browse-acquire-demo;{mode:browse}")
  if linktype == "R":
    rxacquire = session.receiver("browse-acquire-demo")
  else:
    rxacquire = session.receiver("browse-acquire-demo;
{link:{reliability:unreliable}}")

  tx.send("Hello World")

  print "\nBrowser 1 saw message:"
  print msgfetch(rxbrowse1)

  print "Browser 1 then saw message:"
  print msgfetch(rxbrowse1)

  print "\nBrowser 2 saw message:"
  print msgfetch(rxbrowse2)

  print "Browser 2 then saw message:"
  print msgfetch(rxbrowse2)

  print "\nAcquired message:"
  print msgfetch(rxacquire)

  rxacquire.close()

  print "\nBrowser 3 saw message:"
  print msgfetch(rxbrowse3)

except MessagingError, m:
  print m
finally:
  connection.close()

connection.open()
try:
  session=connection.session()

  rxacquire2 = session.receiver("browse-acquire-demo")
  print "\nAcquirer 2 saw message:"
  print msgfetch(rxacquire2)
```

```
except MessagingError, m:
  print m
finally:
  session.acknowledge()
  connection.close()
```

When you select a reliable link for the demonstration, Acquirer 2 sees a redelivered message:

```
Acquirer 2 saw message:
Message(redelivered=True, properties={'x-amqp-0-10.routing-key': u'browse-acquire-
demo'}, content='Hello World')
```

Because the first acquirer did not acknowledge the message acquisition before disconnecting, the broker has returned the message to the queue for redelivery.

When you select an unreliable link for the demonstration, Acquirer 2 does not see any message:

```
Acquirer 2 saw message:
None
```

On an unreliable link, even though the first acquirer did not explicitly accept responsibility for the message by acknowledging acquisition, the broker has deleted the message from the queue. That's the meaning of `unreliable`.

Releasing and Rejecting messages over an unreliable link

It is not possible to release or reject messages acquired over an unreliable link. Over an unreliable link messages are implicitly acknowledged when they are fetched.

Report a bug

5.2.3. Message Rejection

After acquiring a message on a reliable link your application can *reject* it. When a message is rejected the broker will delete it from the queue. If the queue is configured with an `alternate` exchange, then the rejected message is routed there; otherwise it is discarded.

To reject a message using the Python API, call the `acknowledge()` method of the session, passing in the message that you wish to reject, and specify `REJECTED` as the `Disposition` parameter:

Python

```
msg = rx.fetch(timeout = 1)

if msg.content == "something we don't like":
  ssn.acknowledge(msg, Disposition(REJECTED))
else:
  ssn.acknowlege(msg)
```

Note that this is only possible when using a reliable link. When using an `unreliable` link, mesages are implicitly acknowledged when they are fetched.

Report a bug

Prerequisites:

- Section 7.3.2, “Enable Receiver Prefetch”

A Receiver object receives messages from a single subscription. An application can create many receivers, and may wish to deal with messages from these various receivers in the order that the messages are received. The session object provides a method nextReceiver that allows an application to read messages from multiple receivers in a federated order.

Note: To use the Next Receiver feature, prefetch must be enabled for the receivers, and the receivers must be using the same session.

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()
```

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
```

.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();
```

Report a bug

5.2.5. Rejected and Orphaned Messages

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

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

Report a bug

5.2.6. Alternate Exchange

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

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

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

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

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

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

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

Report a bug

Chapter 6. Advanced Queue Features

6.1. Browse-only Queues

Queues declared "browse-only" allow subscribers to access them and acquire their messages normally, but message acquisition transparently results only in a browse. The message will remain on the queue, and accessible to other subscribers.

Messages can only be removed from a browse-only queue by some non-acquisition mechanism: for example, when the message's TTL (time-to-live) duration expires.

The spout and `drain` programs are part of the client libraries package and when installed can be found at:

```
/usr/share/doc/python-qpid-${version}/examples/api/
```

Here is an example of the creation and use of a browse-only queue by the spout and drain clients.

```
./spout \
      -c 10 \
      --broker "localhost:${PORT}" \
      'q; {create: always, node:{type:queue , x-declare:{arguments:{"qpid.browse-
only":1}}}}' \
      "All work and no play makes Mick a dull boy."

      ./drain --broker 'localhost:${PORT}' 'q'
```

See Also:

- Section 5.2, “Browsing and Consuming Messages”

Report a bug

6.2. Ignore Locally Published Messages

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

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

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

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

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

Report a bug

6.3. Exclusive Queues

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

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

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

Report a bug

6.4. Automatically Deleted Queues

6.4.1. Automatically Deleted Queues

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

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

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

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

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

Python

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

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

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

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

Python

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

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

Python

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

An exception to the requirement that a consumer subscribe and then unsubscribe to invoke the auto-deletion is a queue configured to be `exclusive` and `auto-delete`; these queues are deleted by the broker when the session that declared the queue ends, since the session that declared the queue is only possible subscriber.

Report a bug

6.4.2. Automatically Deleted Queue Example

The following Python code demonstrates the behavior of an auto-delete queue. Auto-delete queues are cleaned up by the broker when an application quits. They are usually used to subscribe to an exchange, and a typical use-case is to create an auto-delete queue to specify in the "reply-to" field of a message, to get a response back.

This demonstration uses an auto-delete queue to publish information to a subscriber. This is not a typical use of auto-delete queue, for reasons that we will discover.

Copy the code below and save it as `auto-delete-producer.py`. It can be run using a Python interpreter.

Python

```
import sys
from qpid.messaging import *

connection=Connection("localhost:5672")
connection.open()
try:
  session=connection.session()
  tx=session.sender("test-queue; {create:always, node:{x-declare:{auto-
delete:True}}}")
  tx.send("test message!")
  x = raw_input("Press Enter to continue")
  tx.send("test message 2")
except MessagingError, m:
  print m
connection.close()
```

Restart the broker on the local machine. Whenever the broker is restarted, all non-`durable` queues are deleted. This allows you to start this test with a clean slate.

Run the command:

```
qpid-config queues
```

This lists all the queues on the broker. There will be a dynamically generated queue with a random name with `exclusive` and `auto-del`. This is the queue that `qpid-config` is using to retrieve the list of queues, and will change each time you run the command.

Now start the `auto-delete-producer.py` program using a Python interpreter:

```
python auto-delete-producer.py
```

The program pauses and prompts you to press Enter. Press Enter to continue.

Now run `qpid-config queues` again to list the queues on the broker. This time you will see the `test-queue` that our program created. Our program has exited, but the queue has not been deleted because so far no-one has subscribed to it.

We will now use the `drain` utility to examine the messages on the queue. The `drain` utility is part of the C++ and Python client library packages.

When `drain` runs, it subscribes to the queue, retrieves messages, and then unsubscribes. Run:

```
drain -c 0 test-queue
```

The messages from the `test-queue` will be displayed on the screen. When you run `qpid-config queues` now, you will see that the `test-queue` has been deleted. A consumer subscribed to the queue, and then unsubscribed.

Try the process again, and this time use `drain` to browse the queue, rather than acquire the messages:

```
drain -c 0 "test-queue;{mode:browse}"
```

You will observe that the queue is deleted even when it is browsed. Browsing counts as a subscription as much as acquiring.

Now, to see something very interesting, we will subscribe to the queue and then unsubscribe *while* our program is running.

Copy the following code into a file `auto-delete-subscribe.py`:

Python

```python
import sys
from qpid.messaging import *

connection=Connection("localhost:5672")
connection.open()
try:
  session=connection.session()
  rx=session.receiver("test-queue")
  print rx.fetch(timeout = 1)
  session.acknowledge()
except MessagingError,m:
  print m
connection.close()
```

Now run `auto-delete-producer.py`. When it pauses, run `auto-delete-subscriber.py`, then check `qpid-config queues`. You'll see that the queue has been deleted.

Now press Enter to continue. When the program finishes, use `drain` to browse the `test-queue`. It doesn't exist.

The `test-queue` created by `auto-delete-producer.py` was deleted when our consumer program subscribed to the queue by creating and attaching a receiver, and then unsubscribed by closing the connection. The second message sent by our message producer was never delivered and *no exception was raised*.

This is something to be aware of: a sender is a handle to a local router that routes messages to the message broker. The constructor parameter of the sender is a routing key. Our constructor is the name of a queue, but a sender always routes messages to an exchange. When no exchange is specified, the default exchange is used: a nameless direct exchange on the broker. The sender's constructor checks that the routing key it is given refers to a valid target on the message broker, so it checks that there is a "test-queue" on the default exchange. At the time the sender is created this queue exists. After that, the sender's send method routes messages to the default exchange on the broker with a routing key set to "test-queue". Since the target exchange still exists no exception is raised when we send. The message arrives at the default exchange on the broker, where it is discarded because there is no queue subscribed to the exchange that matches the routing key.

To avoid this scenario, you should either use a non-auto-deleting queue for publishing, or you can create and subscribe a receiver alongside the sender. This guarantees that the queue will continue to exist for the lifetime of your sender. To do this in our program, we will create and subscribe a receiver directly after the sender creates the queue. We will also add a second pause where we can check the existence and state of the `test-queue`. Here's the updated program:

Python

```
import sys
from qpid.messaging import *

connection=Connection("localhost:5672")
connection.open()
try:
  session=connection.session()
  tx=session.sender("test-queue; {create:always, node:{x-declare:{auto-
delete:True}}}")
  rx=session.receiver("test-queue")
  tx.send("test message!")
  x = raw_input("Press Enter to continue")
  tx.send("test message 2")
  x = raw_input("Press Enter to continue")
except MessagingError, m:
  print m
connection.close()
```

Now start the `auto-delete-producer.py` program. Run `auto-delete-subscriber.py` in the first pause. Previously, this would delete the queue, and the second message would go nowhere. This time our producer's own subscription is keeping the queue alive. Press Enter to have `auto-delete-producer.py` send the second message. Now check the queue using either `drain` or `auto-delete-subscriber.py`. This time you'll see that the queue exists and the message has been delivered as expected.

Report a bug

6.4.3. Queue Deletion Checks

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

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

Report a bug

6.5. Last Value (LV) Queues

6.5.1. Last Value Queues

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

Report a bug

6.5.2. Declaring a Last Value Queue

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

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

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

To create the same queue in an application:

Python

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

Report a bug

6.5.3. Last Value Queue Example

This example demonstrates how to create and use a Last Value Queue. The language bindings and programming details differ between languages, but the principles are the same.

We will create a messaging queue that provides regular stock price updates. Message consumers are interested in the current stock price, and do not wish or need to receive messages with historical information. A last value queue is perfect for this application: newly arriving messages can update and replace older ones.

We will call our queue "stock-ticker". Our stock-ticker queue will use "stock-symbol" as the last value queue key. The value of this key in the message header will identify a message as a new message to the queue, or an update to a message already in the queue.

First we import the Qpid Messaging client library:

Python

```
import sys
from qpid.messaging import *
```

Now we create a Connection to the broker running on the standard AMQP port, 5672, on the local machine:

Python

```
connection = Connection("localhost:5672")
connection.open()
```

And now we use this connection to create a session:

Python

```
session = connection.session()
```

Now we create a sender and declare a last value queue at the same time. We will create a queue called "stock-ticker", and use "stock-symbol" as the last value queue key. Messages sent to this queue will identify themselves as an update to a previous message by specifying the same "stock-symbol" in their headers.

The following statement is a single line of code. It may break across lines in display, but it should be entered as a single line.

Python

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

Sidenote: We could also create the queue using the `qpid-config` command line tool:

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

Now let's create and send some messages to the queue. We use the "stock-symbol" key in the header to identify which stock a message describes. Our last value queue uses this header key to match our message with messages already in the queue.

Python

```
msg1 = Message("10")
msg1.properties = {'stock-symbol':'RHT'}

msg2 = Message("10")
msg2.properties = {'stock-symbol':'JAVA'}

msg3 = Message("10")
msg3.properties = {'stock-symbol':'MSFT'}

msg4 = Message("12")
msg4.properties = {'stock-symbol':'RHT'}
```

After sending these messages to our last value queue a new consumer should see three messages in the queue, one for each stock symbol, with `msg4` updating `msg1`. To contrast the behavior of the last value queue with a standard FIFO queue, we'll send our messages to a control queue, called *control-queue* at the same time:

Python

```
controlSender = session.sender("control-queue;{create:always,
node:{type:queue}}")
```

Now we send our messages to the two queues:

Python

```
stockSender.send(msg1)
controlSender.send(msg1)

stockSender.send(msg2)
controlSender.send(msg2)

stockSender.send(msg3)
controlSender.send(msg3)

stockSender.send(msg4)
controlSender.send(msg4)
```

Our messages are now in the queues. We create two receivers to now examine the content of the queues:

Python

```
stockBrowser = session.receiver("stock-ticker; {mode:browse}")
controlBrowser = session.receiver("control-queue; {mode:browse}")
```

These are browsing receivers, so they do not acquire messages and remove them from the queue. To clear the queues, remove the browse property from the receiver declarations, like so: `session.receiver("stock-ticker")`, and run the demo again. With the receivers browsing, you will be able to see more distinctly the effect of a Last Value Queue over time by running the demo several times in succession without clearing the queues.

We will use the prefetch capability of the receivers to browse messages on the queue, and to allow us to count how many messages are in the queue using the `available()` method. We do this by setting the receivers' prefetch `capacity` to a value higher than the default of 0:

Python

```
stockBrowser.capacity = 20
controlBrowser.capacity = 20
```

Once the prefetch capacity of the receiver is set to 20, up to 20 available messages are retrieved asynchronously from the queue. Because the operation is asynchronous we need to wait for it to complete. We will put our application to sleep for 10 seconds before examining the prefetched messages:

Python

```
sleep 10
```

We need to import `sleep` from the time library:

Python

```
from time import sleep
```

Note that we do this in order to examine the `available()` property of the receiver with certainty that this represents the number of messages in the queue. When operating asynchronously `available()` reports the number of messages available locally. After a ten second delay, we can be reasonably certain that this is the total number of messages in the queue. In an actual asynchronous operation you would not want to block execution of your application. Instead you would use a pattern like this:

Python

```python
while True:
  try:
    msg = stockBrowser.fetch(timeout = 10)
    print msg.properties["stock-symbol"] + ":" + msg.content
  except Empty:
    break
```

When our application finishes its sleep cycle, we will examine the number of messages in the queue, and print them out:

Python

```python
print "Last Value Queue has " + str(stockBrowser.available()) + " messages"

print "\nLast Value Queue messages:"

for x in range(stockBrowser.available()):
  try:
    msg = stockBrowser.fetch(timeout = 1)
    print msg.properties["stock-symbol"] + ":" + msg.content
  except MessagingError, m:
    pass

print "Control Queue has " + str(controlBrowser.available()) + " messages"

print "\nControl Queue messages:"
for x in range(controlBrowser.available()):
  try:
    msg = controlBrowser.fetch(timeout = 1)
    print msg.properties["stock-symbol"] + ":" + msg.content
  except MessagingError, m:
    pass
```

And finally we acknowledge our session and close the connection:

Python

```python
session.acknowledge()
connection.close()
```

We are now ready to run our test. Here's the complete program listing:

Python

```
import sys
from qpid.messaging import *
from time import sleep

connection = Connection("localhost:5672")
try:
  connection.open()
  session = connection.session()

  stockSender = session.sender("stock-ticker;{create:always, node:{type:queue,
x-declare:{arguments:{'qpid.last_value_queue_key': 'stock-symbol'}}}}")
  controlSender = session.sender("control-queue;{create:always,
node:{type:queue}}")

  stockBrowser = session.receiver("stock-ticker;{mode:browse}")
  controlBrowser = session.receiver("control-queue;{mode:browse}")
  controlBrowser = session.receiver("control-queue")

  msg1 = Message("10")
  msg1.properties = {'stock-symbol':'RHT'}

  msg2 = Message("10")
  msg2.properties = {'stock-symbol':'JAVA'}

  msg3 = Message("10")
  msg3.properties = {'stock-symbol':'MSFT'}

  msg4 = Message("12")
  msg4.properties = {'stock-symbol':'RHT'}

  stockSender.send(msg1)
  controlSender.send(msg1)

  stockSender.send(msg2)
  controlSender.send(msg2)

  stockSender.send(msg3)
  controlSender.send(msg3)

  stockSender.send(msg4)
  controlSender.send(msg4)

  stockBrowser.capacity = 20
  controlBrowser.capacity = 20

  sleep(10)

  print "\nLast Value Queue has " + str(stockBrowser.available()) + " messages"

  print "Last Value Queue messages:"

  for x in range(stockBrowser.available()):
    try:
      msg = stockBrowser.fetch(timeout = 1)
      print msg.properties["stock-symbol"] + ":" + msg.content
    except MessagingError, m:
      pass

  print "\nControl Queue has " + str(controlBrowser.available()) + " messages"

  print "Control Queue messages:"

    for x in range(controlBrowser.available()):
    try:
      msg = controlBrowser.fetch(timeout = 1)
      print msg.properties["stock-symbol"] + ":" + msg.content
    except MessagingError, m:
      pass

  session.acknowledge()

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

Report a bug

6.5.4. Last Value Queue Command-line Example

The included programs drain and spout can be used for sending and receiving messages for testing purposes. The source code for the two utilities is included in the Python and C++ client library packages. The Python version can be run uncompiled using a Python interpreter.

Run the following qpid-config command to create a Last Value Queue:

```
qpid-config add queue my-queue --argument qpid.last_value_queue_key=type
```

The header key 'type' is used to match messages in the queue.

Now start one or more browsers using the drain command:

```
./drain -f -c 0 'my-queue; {mode: browse}'
```

These browsers will see all the messages as they arrive in the queue in real-time.

Now use spout to send messages to the queue, setting a header value for the key 'type':

```
./spout -P type=a my-queue a1
./spout -P type=a my-queue a2
./spout -P type=a my-queue a3
./spout -P type=b my-queue b1
./spout -P type=c my-queue c1
./spout -P type=c my-queue c2
./spout -P type=a my-queue a4
```

The browsers started before these messages were published will see all messages as they arrive.

Now start a new browser:

```
./drain -c 0 'my-queue; {mode: browse}'
```

This browser will see only the last messages for each of the unique 'type' values.

Report a bug

6.6. Priority Queuing

6.6.1. Priority Queuing

Priority queues deliver messages based on their priority. Higher priority messages are delivered before lower priority messages. A total of 10 distinct priority levels are possible.

A priority queue is declared with a `qpid.priority` attribute. This attribute is an integer value between 1 and 10, and defines the number of distinct priority levels for the queue.

For example, when the `qpid.priority` attribute of a queue is set to 10, there are ten distinct priority levels for the queue. In this case a message with a priority level of 10 is delivered before a message with a priority of 9, which is delivered before a message with a priority level of 5, which is delivered before a message with a priority level of 1.

If the `qpid.priority` attribute of a queue is set to 2, there are two distinct priority levels for the queue. In this case message priorities 6-10 is the queue priority level 1, and message priorities 1-5 is the queue priority level 2. Messages in the same priority band are delivered based on their priority and the order in which they are received.

Report a bug

6.6.2. Declaring a Priority Queue

To declare a priority queue, specify a value for `qpid.priorities` in the `x-declare` arguments of the node declaration. For example:

Python

```
sender = session.sender('my-queue; {create: always, node:{x-
declare:{arguments:{qpid.priorities:10}}}}')
```

Using `qpid-config`:

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

Report a bug

6.6.3. Considerations when using Priority Queues

Browsing Consumers and Priority Queues

Priority Queues deliver messages to acquiring consumers in order of priority, rather than the usual First-In-First-Out (FIFO) order of a queue. The delivery order for *browsing consumers* is "undefined". At the time of writing, browsing consumers receive messages from a priority queue in FIFO order; however, you should not rely on this behavior in your applications, as it may change in the future.

Fairshare feature

If the message enqueue rate sufficient outpaces the dequeue rate in a priority queue, it is possible that lower priority messages may never be removed from the queue. To avoid this situation the *Fairshare feature* allows a consumer to take a specified block of message from each priority level in turn.

Report a bug

6.6.4. Priority Queue Demonstration

The following program demonstrates the use and behavior of a priority queue.

Python

```
#!/usr/bin/python

import sys
from qpid.messaging import *

connection = Connection("localhost:5672")
connection.open()
try:
  ssn = connection.session()

  x = 0
  print "\n"
  while True:
    print "Create queue with 2 or 10 priority levels?"
    x = raw_input()
    if (x == "2") or (x == "10"):
      break

  tx = ssn.sender("nonpriority-demo-queue; {create: always, node: {type:
'queue'}}")
  print "Creating a priority queue with " + x + " priority levels:"
  address =  "priority-demo-queue; {create: always, "
  address = address + "node:{x-declare: {auto-delete:True, "
  address = address + "arguments: {'qpid.priorities': "
  address = address + x + "}}}}"
  print address
  txpriority = ssn.sender(address)

  rx = ssn.receiver('nonpriority-demo-queue')
  rxpriority = ssn.receiver("priority-demo-queue")
  rxbrowse = ssn.receiver("priority-demo-queue; {mode: browse}")

  print "\nPress Enter to continue\n"
  x = raw_input()

  print "First message sent:"
  msg = Message("priority 1")
  msg.priority = 1
  tx.send(msg)
  txpriority.send(msg)
  print msg

  print "Second message sent:"
  msg = Message('priority 4')
  msg.priority = 4
  tx.send(msg)
  txpriority.send(msg)
  print msg

  print "\nPress Enter to continue\n"
  x = raw_input()
  print "BROWSE PRIORITY QUEUE"
  print "First browse in priority queue:"
  print rxbrowse.fetch()

  print "Second browse in priority queue:"
  print rxbrowse.fetch()

  print "\nPress Enter to continue\n"
  x = raw_input()

  print "ACQUIRE PRIORITY QUEUE"
  print "First message in priority queue:"
  print rxpriority.fetch()
```

```
    print "Second message in priority queue:"
    print rxpriority.fetch()

    print "\nPress Enter to continue\n"
    x = raw_input()

    print "ACQUIRE NON-PRIORITY QUEUE"
    print "First message in non-priority queue:"
    print rx.fetch()

    print "Second message in non-priority queue:"
    print rx.fetch()

    ssn.acknowledge()
  finally:
    connection.close()
```

When run, this program allows you to create a priority queue with 2 or 10 priority levels. It then sends two messages to this queue, with priorities 1 and 4. It then demonstrates the behavior of browsing and acquiring from the priority queue, and contrasts this with acquiring from a non-priority queue.

Here is the output when the program is run and a priority queue with 10 distinct priority levels is created:

```
Create queue with 2 or 10 priority levels?
10
Creating a priority queue with 10 priority levels:
priority-demo-queue; {create: always, node:{x-declare: {auto-delete:True,
arguments: {'qpid.priorities': 10}}}}
```

The queue is declared as `auto-delete: True` to allow the program to be run multiple times with different values for `qpid.priorities`. If the queue already exists when the sender is created, the value given for `qpid.priorities` has no effect. This value only has an effect when the queue is created.

```
First message sent:
Message(priority=1, content='priority 1')
Second message sent:
Message(priority=4, content='priority 4')
```

Two messages are sent, one with priority 1 (the lowest priority), and one with priority 4 (a higher priority).

The first examination is of a browsing receiver. Priority queuing has no effect for browsers, only acquiring consumers, so we see our messages in the order they were sent - FIFO *First In, First Out*:

```
BROWSE PRIORITY QUEUE
First browse in priority queue:
Message(priority=1, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 1')
Second browse in priority queue:
Message(priority=4, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 4')
```

However, when we acquire the messages from the priority queue, we see that they are dequeued in order of descending priority - our priority 4 message is delivered before the priority 1 message, even though it was sent later:

```
ACQUIRE PRIORITY QUEUE
First message in priority queue:
Message(priority=4, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 4')
Second message in priority queue:
Message(priority=1, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 1')
```

Finally, for contrast, the messages are dequeued from a non-priority queue, where they are delivered in the order they were received by the broker:

```
ACQUIRE NON-PRIORITY QUEUE
First message in non-priority queue:
Message(priority=1, properties={'x-amqp-0-10.routing-key': u'nonpriority-demo-
queue'}, content='priority 1')
Second message in non-priority queue:
Message(priority=4, properties={'x-amqp-0-10.routing-key': u'nonpriority-demo-
queue'}, content='priority 4')
```

When the demonstration is run and a priority queue with only 2 distinct levels is select, you will observe that the priority queue delivers the message in the same order they were delivered:

```
Create queue with 2 or 10 priority levels?
2
Creating a priority queue with 2 priority levels:
priority-demo-queue; {create: always, node:{x-declare: {auto-delete:True,
arguments: {'qpid.priorities': 2}}}}

....

ACQUIRE PRIORITY QUEUE
First message in priority queue:
Message(priority=1, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 1')
Second message in priority queue:
Message(priority=4, properties={'x-amqp-0-10.routing-key': u'priority-demo-
queue'}, content='priority 4')
```

When a queue has only two distinct priority levels, those levels are the message priority bands 1-5 and 6-10. Since our messages both have priorities in the band 1-5, they are considered to have the same priority, and are delivered based on the order they were received by the broker.

Report a bug

6.6.5. Fairshare Feature

When using a priority queue, a velocity mismatch between message producers and consumers can result in lower priority messages remaining in the queue indefinitely. To ensure that messages of all priorities are serviced, the *fairshare* feature can be used to grab a predetermined number of messages for each priority level.

The `x-qpid-fairshare` argument of `x-declare: argument` can be used to enforce either a common number of messages to be grabbed per-priority-level, or a custom number of messages per-priority-level

The following example creates a queue with 10 priority levels, and will grab 5 messages from each priority in turn:

```
qpid-config add queue 'my-queue; {create: always, node:{x-
declare:{arguments:{qpid.priorities:10, x-qpid-fairshare: 5}}}}'
```

The following example creates a queue with 10 priority levels, with custom fairshare amounts per-priority-level:

```
qpid-config add queue 'my-queue; {create: always, node:{x-
declare:{arguments:{qpid.priorities:10, x-qpid-fairshare-0: 3, x-qpid-fairshare-1:
5, x-qpid-fairshare-2: 3, x-qpid-fairshare-3: 2, x-qpid-fairshare-4: 4, x-qpid-
fairshare-5: 5, x-qpid-fairshare-6: 5, x-qpid-fairshare-7: 3, x-qpid-fairshare-8:
5, x-qpid-fairshare-9: 4, x-qpid-priorities: 10}}}}'
```

Report a bug

6.7. Message Groups

6.7.1. Message Groups

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

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

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

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

Report a bug

6.7.2. Create a Queue with Message Groups enabled

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

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

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

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

Python

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

C++

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

Report a bug

6.7.3. Message Group Consumer Requirements

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

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

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

Report a bug

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

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

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

Report a bug

6.7.5. Create a Queue with Message Groups enabled

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

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

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

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

Python

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

C++

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

Report a bug

6.7.6. Default Group

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

Report a bug

6.7.7. Override the Default Group Name

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

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

Report a bug

6.7.8. Message Groups Demonstration

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

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

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

Python

```
import sys
from qpid.messaging import *

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

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

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

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

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

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

  counter +=1
  return msg

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

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

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

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

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

    print "\nMessage grouping is disabled"
    msggroup = False

  else:

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

    print "\nMessage grouping is enabled"
    msggroup = True

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

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

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

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

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

  counter = 0
  pullmsg(1)
  pullmsg(2)

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

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

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

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

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

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

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

    print "\nPulling more messages from the queue:"

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

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

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

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

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

Example program output

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

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

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

Queued message: A1
Consumer 1 got: A1

Queued message: B1
Consumer 2 got: B1

Queued message: B2
Consumer 1 got: B2

Queued message: A2
Consumer 2 got: A2

Queued message: B3
Consumer 1 got: B3

Queued message: A3
Consumer 2 got: A3

Queued message: B4
Consumer 1 got: B4

Queued message: B5
Consumer 2 got: B5

Queued message: A4
Consumer 1 got: A4

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

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

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

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

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

Queued message: A1
Consumer 1 got: A1

Queued message: B1
Consumer 2 got: B1

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

Queued message: B2
Consumer 1 got: A2

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

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

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

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

Pulling more messages from the queue:

Queued message: B3
Consumer 1 got: B3

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

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

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

Sending some more messages to the queue...

Queued message: B4
Consumer 1 got: B4

Queued message: B5
Consumer 2 got: A4

Queued message: A4
Consumer 1 got: B5

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

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

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

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

Pulling more messages from the queue:

Queued message: B3
Consumer 1 got: A3

Queued message: A3
Consumer 2 got: B3

Sending some more messages to the queue...

Queued message: B4
Consumer 1 got: A4

Queued message: B5
Consumer 2 got: B4

Queued message: A4
Consumer 1 got: A5

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

Report a bug

Chapter 7. Asynchronous Messaging

7.1. Asynchronous Operations

Asynchronous operations allows some communication with the broker to take place in the background, while your program continues to execute. When send and receive operations are performed synchronously execution is blocked while communication takes place between the client and the broker.

Asynchronous send allow execution to continue without waiting on acknowledgement from the server. Asynchronous receive enables receivers to retrieve messages in the background, so that when you wish to retrieve a message using a receiver in your code, the message has already been fetched and is available locally.

Asynchronous operations significantly improve throughput; but you should understand the behavior of asynchronous operations and carefully manage it in your code.

Report a bug

7.2. Asynchronous Sending

7.2.1. Synchronous and Asynchronous Send

When a sender sends synchronously over a reliable link, execution in the sender's thread is blocked until the sender receives an acknowledgement from the broker. This is useful for testing and troubleshooting, but by introducing a round-trip for every message, this reduces the potential throughput of the system.

When using the C++ API, all calls are *asynchronous* by default. When using the Python API, however, the opposite is true - by default, a sender sends a message synchronously.

You can send messages asynchronously, which allows you to maximise your network bandwidth usage and throughput. When invoked asynchronously, a send call will return immediately, without waiting for a receipt from the broker.

For example, the following call to the `send()` method of a send object is asynchronous - it returns immediately, without waiting for a receipt from the broker:

Python

```
sender.send(message, sync = False)
```

C++

```
sender.send(message, false)
```

Note that this is the default behavior for the C++ API.

Report a bug

7.2.2. Sender Capacity

Sender `capacity` is the property of a sender object that controls the number of asynchronous sends pending acknowledgement from the server that the sender will permit. These unacknowledged messages are buffered in memory for retransmission in the event of a link failure, so the sender capacity is also known as the sender *replay buffer size*.

By default, sender capacity is set to `UNLIMITED`, meaning that the sender will allow an unlimited number of asynchronous calls to be made, and buffer a number of messages that is limited only by the memory limits of the system.

When the sender `capacity` is set to a number other than UNLIMITED, the sender will allow only that many asynchronous send operations to be outstanding at the same time.

For example: if a sender's `capacity` is set to 10, then a maximum of 10 asynchronous send operations can be awaiting acknowledgement at the same time for the sender. If 10 asynchronous send operations are invoked, and an 11th operation is attempted before any of those 10 are acknowledged by the broker, then the sender will block until one of the asynchronous send operations is acknowledged by the broker.

Be aware of two things: unbounded sender capacity can have an impact on resources if your sender outpaces the server significantly. Be aware also that upon reaching its capacity a sender will switch from asynchronous to synchronous send behavior, and message sends will block. You should tune your sender capacity with this in mind, and also carefully program your send operations to check the sender's capacity and availability if blocking will be problematic.

Report a bug

7.2.3. Set Sender Capacity

In Python, the sender capacity is set by assigning a value to the `capacity` property of a sender. In C++, the sender capacity is set using the `setCapacity` method.

Python

```
sender.capacity = 20
```

C++

```
sender.setCapacity(20)
```

Report a bug

7.2.4. Query Sender Capacity

When using asynchronous message sending, three sender properties are available to ascertain the state of the asynchronous calls. They are:

Sender Capacity

The maximum number of asynchronously sent messages that can be pending acknowledgement at any given time. By default this is `UNLIMITED`, but it can be changed to constrain the number of unsettled asynchronous calls. An attempt to make a further asynchronous call when the sender is at capacity will block until another sent message is acknowledged by the broker.

C++

```
sender.getCapacity()
```

Python

```
sender.capacity
```

Sender Unsettled

The number of asynchronous sends pending acknowledgement from the broker.

C++

```
sender.getUnsettled()
```

Python

```
sender.unsettled()
```

Sender Available

The number of additional asynchronous calls that the sender can accept at the moment. This value is available as a property, but can also be computed from `sender.capacity` - `sender.unsettled`.

C++

```
sender.getAvailable()
```

Python

```
sender.available()
```

Report a bug

7.2.5. Avoiding a Blocked Asynchronous Send

An asynchronous send call will place the message into the send buffer and return to execution immediately. However, if the send buffer is full the call will block until space is available.

If you need to ensure that an asynchronous send call does not block on a full buffer, you should query the buffer state before making the call. For example, in C++:

C++

```
  if (sender.getAvailable() > 0)
      sender.send(message, false)
   // else drop the message
```

Python

```
if sender.available() > 0:
  sender.send(message, sync=False)
else:
  # drop the message
```

You can also increase the size of the sender's replay buffer to reduce the chances of it filling up:

C++

```
sender.setCapacity(SOME_LARGE_NUMBER)
```

Python

```
sender.capacity = SOME_LARGE_NUMBER
```

Report a bug

7.2.6. Asynchronous Message Sending Example

The following code demonstrates using the properties of a sender to manage asynchronous send operations, with the option to avoid synchronous blocking when the sender is at capacity:

C++

```
sender.setCapacity(MY_CAPACITY);

// Later
bool resend = true;
while (resend)
{
  if (sender.getAvailable()>0)
  {
    sender.send(message,false);
    resend = false;
  }
  else
  {
    // May wish to do nothing here
    // or send to log file
    std::cout << "Warning: Capacity \ full. Retry" << std::endl;
   }
}
// Later
if (sender.getUnsettled())
{
    session.sync();
}
```

Python

```
snd.capacity = MY_CAPACITY

# Later

resend = True
while (resend):
  if (snd.available()>0):
    snd.send(msg, sync = False)
    resend = False
  else:
    print "Warning: Capacity full"

# Later
    if (snd.unsettled()):
      ssn.sync()
```

Report a bug

7.2.7. Asynchronous Send and Link Reliability

The `sender.capacity` is the number of unacknowledged sends that a sender will allow when sending asynchronously. The two-phase send/acknowledge behavior is a characteristic of a reliable link (technically known as a link with *at-least-once* reliability). The sender sends a message, and buffers that message locally until the server responds to acknowledge receipt of the message. This buffering of unacknowledged sent messages enables the sender to resend messages (*sender replay*) if the link is dropped and then re-established. When a reliable link is dropped and then transparently re-established, messages that were sent asynchronously but not acknowledged by the server are resent from the sender replay buffer.

A reliable link is the default link used when creating a sender with no explicit link reliability specified. You can explicitly request an `unreliable` link when creating a sender. For example:

Python

```
sender = session.sender("amq.topic;{link: {'reliability': 'unreliable'}}")
```

When using an `unreliable` link, sender capacity has no meaning. On an unreliable link the server does not acknowledge receipt of messages. All messages are considered as good as acknowledge once they are sent. This is the meaning of `unreliable` for a sender. If the link is dropped there is no way for the sender to know which messages made it to the broker and which were lost. This also means that over an unreliable link asynchronous senders will not block, as their capacity is never utilized.

`Sender.capacity` is used to limit the exposure of an application to data loss, and the amount of memory that senders can consume with their replay buffer. It can also be used to throttle producers. You can use an unreliable link along with asynchronous send to maximise throughput without the implications of local memory required for the sender replay buffer, and no throttling of producers. However, you must be aware of the reduced reliability and employ this pattern in situations where the potential for data loss is not important.

The following program demonstrates the difference between asynchronous sending over reliable and unreliable links:

Python

```
import sys
from qpid.messaging import *

connection = Connection("localhost:5672")

try:
  connection.open()
  session = connection.session()

  linktype=""
  while linktype != "R" and linktype !="U":
    response = raw_input("Use (R)eliable or (U)nreliable link [R/U]? ")
    linktype = response.upper()

  if linktype == "U":
    sender = session.sender("amq.topic;{link: {'reliability': 'unreliable'}}")
  else:
    sender = session.sender("amq.topic")

  message = Message("Hello World:")
  print sender.capacity
  sender.capacity = 5
  for x in range (1000):
    if sender.available() == 0:
      print "Sender is blocking..."
    sender.send("Hello World: " + str(x), sync=False)
    print str(x) +" : " + str(sender.unsettled()) + " : " +
str(sender.available())

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

The program sends 1000 messages asynchronously over a link using a sender with a capacity of 5 unacknowledged messages. The output is:

```
message number : unacknowledged messages : further async send capacity
```

When run over a reliable link you will see the number of unacknowledged messages and the remaining async send capacity vary, including occasions where the asynchronous sender will block:

```
Use (R)eliable or (U)nreliable link [R/U]? R
...
918 : 1 : 4
919 : 2 : 3
920 : 3 : 2
921 : 4 : 1
922 : 5 : 0
Sender is blocking...
```

You can experiment with the value for `sender.capacity` (set to 5 in the program code) to see the impact it has on sender blocking.

When run over an unreliable link, you will see that `sender.capacity` has no impact on the performance of the sender. Remember, however, that it is now unreliable:

```
Use (R)eliable or (U)nreliable link [R/U]? U
...
984 : 0 : 5
985 : 0 : 5
986 : 0 : 5
987 : 0 : 5
988 : 0 : 5
989 : 0 : 5
```

Report a bug

7.3. Asynchronous Receiving

7.3.1. Asynchronous Message Retrieval (Prefetch)

By default, a receiver retrieves a single message synchronously in response to a `fetch()` call. The receiver's capacity to *prefetch* messages is 0 by default.

When the receiver's capacity is set to a value greater than 0, the receiver will asynchronously retrieve up to that number of messages from the queue. This asynchronous retrieval is called *prefetch*, and it is enabled and controlled by setting the `capacity` property of a receiver.

Prefetching messages has two advantages:

- Prefetched messages are available locally when requested by the application, without the overhead of a synchronous call to retrieve a message from the broker.
- A receiver with prefetching enabled has an `available()` method that can be invoked to determine how many prefetched messages are available.

Note two things about the `available()` method:

Prefetching is asynchronous, which means that you cannot rely on the number returned by a call to `available()` as an absolute indicator of the state of the queue. For example, calling `available()` immediately after setting the capacity of a receiver to something other than 0 is likely to return a value of 0 messages available. This does not necessarily mean that the queue has no messages, but rather than no pre-fetched messages are locally available yet.

Note also that the maximum value reported by the `available` method of a receiver with prefetching enabled will be the `capacity` of the receiver. The `available()` method reports the number of prefetched messages available, not the number of messages in the queue. If the number of available messages is less than the capacity of the receiver, however, you can infer that this is the number of messages in the queue, with the above caveat about the asynchronous nature of prefetching.

Report a bug

7.3.2. Enable Receiver Prefetch

To enable a receiver to prefetch messages, set its capacity to a value greater than 0.

For example, the following code creates a receiver and enables prefetching of up to 100 messages:

Python

```
import sys
from qpid.messaging import *

connection = Connection("localhost:5672")
connection.open()
ssn = connection.session()

prefetchingReceiver = ssn.receiver("testqueue; {create:always}");
prefetchingReceiver.capacity = 100
```

Report a bug

7.3.3. Asynchronously Acknowledging Received Messages

A reliable link (technically called a link with *at-least-once* reliablity) is the default link used when a receiver is created without specifying a link reliability. For message acknowledgement on unreliable links refer to Acknowledging Messages Received Over an Unreliable Link. Messages received over a reliable link are set to `acquired` on the broker until they are acknowledged by the consumer. When a message is in `acquired` mode it is not visible in the queue. If the consumer disconnects without acknowledging receipt, the message will be moved out of `acquired` and again become available to consumers, with the header `redelivered=true`.

To remove the message from the queue, a consumer needs to acknowledge receipt of the message.

In Python, this is done by calling the `acknowledge()` method of the `session` object:

Python

```
session.acknowledge()
```

Calling the `acknowledge()` method with no arguments acknowledges receipt of all as-yet-unacknowledged messages fetched using that session. To acknowledge a specific message, pass the message as an argument. For example:

Python

```
msg = rx.fetch(timeout = 1)
session.acknowledge(msg)
```

This method executes synchronously by default, and will wait for the broker to respond before returning. It can also be invoked asynchronously, by supplying the `sync = False` parameter:

Python

```
session.acknowledge(msg, sync = False)
```

Acknowledging Messages Received Over an Unreliable Link

When an `unreliable` link is requested for a receiver, acknowledgement is implicit when a message is fetched. This means that the broker marks the message as acquired as soon as the receiver fetches it. No acknowledgement is necessary, and no release or rejection of messages is possible.

Report a bug

7.3.4. Asynchronous Receive and Link Reliability

Bear in mind that the combination of asynchronous receive (prefetch) and an `unreliable` link is a potentially lossy situation. Over an `unreliable` link, when an application is consuming (as opposed to browsing the queue) the broker deletes the message from the queue as soon as it is prefetched. It does not wait for acknowledgement from the consumer. If the consumer fails before it dispatches prefetched messages, the broker will not redeliver them.

When using this combination - asynchronous receive (prefetch) and `unreliable` link - be aware of the implications.

Report a bug

Chapter 8. Reliability and Quality of Service

8.1. Link Reliability

8.1.1. Reliable Link

The link established when connecting to a queue is *reliable* by default. Technically, this is *at-least-once* reliability.

Receiving messages over a reliable link

An acquiring message consumer (also known as a *competing message consumer*) is a message consumer who removes messages from a queue, and makes them unavailable to other consumers. When an acquiring message consumer fetches a message from the broker over a reliable link, the message is set to `acquired`. In the acquired state the message is not visible to other consumers. It is to all intents and purposes acquired by the consumer, but the broker maintains its copy in acquired state until the consumer *acknowledges* acquisition. At that point the broker considers the message reliably delivered, and will delete its copy.

The reliable link enables several behaviors. If a consumer closes its connection to the server without acknowledging the message, the broker will assume that the consumer has failed. In this case the acquired message is returned to the queue, with a header `redelivered: true`.

Additionally, the consumer may choose to explicitly *release* the message, in which case the broker will perform the same action; or the consumer may choose to *reject* the message. When a message is rejected, the broker will route the message to the `alternate exchange`, if one has been configured for this queue or exchange. If no `alternate exchange` is configured, the message will be discarded.

Sending messages over a reliable link

When a message is sent to the broker over a reliable link, the sender maintains its local copy until the broker acknowledges receipt. At that time the sender deletes the local copy. When sending synchronously this causes the application to block until this exchange has taken place. When sending asynchronously these unacknowledged sent messages are stored in the *sender replay buffer*.

When a reliable link is dropped momentarily and then re-established, the sender will resend unacknowledged messages from its buffer, ensuring that no data is lost. This may result in messages being sent more than once, hence the term at-*least*-once.

Specifying a reliable link

All links to queues are reliable by default. It is not necessary to explicitly request a reliable link when connecting to a queue.

When connecting to an exchange the link is unreliable by default. To specify a reliable link to an exchange, include `link: {'reliability': 'at-least-once'}` in the address. For example:

```
sender = session.sender("amq.topic;{link: {'reliability': 'at-least-once'}}")
```

In this case, the sender will follow the reliable link behavior, buffering messages locally until they are acknowledged by the broker.

Report a bug

8.1.2. Unreliable Link

The link established when connecting to an exchange is *unreliable* by default. Additionally, an application can explicitly request an `unreliable` link when establishing a connection to a queue.

An unreliable link sends data fast and loose. There is no buffering either on the server or on the local client to guard against lost connections. When a client takes a message from a queue over an `unreliable` link, the broker deletes it immediately, without waiting for the consumer to acknowledge that it received and successfully actioned a message.

In some scenarios you may see an increase in throughput when using an `unreliable` link, although this is be no means certain. The most obvious use for an unreliable link is when a large volume of data is being transmitted at high speed and data loss is not an issue.

Most applications benefit from the guarantees provided by the reliable link, and it is the default for all links.

Requesting an `unreliable` link

To request an `unreliable` link, specify `link: {'reliability': 'unreliable'}` in the address for the receiver or sender. For example:

Python

```
sender = session.sender("amq.topic;{link: {'reliability': 'unreliable'}}")
```

Report a bug

8.2. Queue Sizing

8.2.1. Controlling Queue Size

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

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

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

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

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

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

Python

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

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

Behavior when limits are reached: `qpid.policy_type`

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

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

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

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

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

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

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

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

The same thing is achieved in an application like so:

Python

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

See Also:

- Section 8.3, “Producer Flow Control”

Report a bug

8.2.2. Queue Threshold Alerts

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

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

The following code demonstrates subscribing to and consuming alert messages:

Python

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

Alert Repeat Gap

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

Backwards-compatible aliases

The following aliases are maintained for compatibility with earlier clients:

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

Report a bug

8.3. Producer Flow Control

8.3.1. Flow Control

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

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

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

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

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

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

When the resource utilization of the queue falls below the `flow_resume_threshold`, producer flow control is stopped. Setting the `flow_resume_threshold` above the `flow_stop_threshold` has the obvious consequence of locking producer flow control on, so don't do it.

Report a bug

8.3.2. Queue Flow State

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

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

Report a bug

8.3.3. Broker Default Flow Thresholds

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

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

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

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

Report a bug

8.3.4. Disable Broker-wide Default Flow Thresholds

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

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

Report a bug

8.3.5. Per-Queue Flow Thresholds

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

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

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

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

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

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

Report a bug

8.4. Credit-based Flow Control

8.4.1. Flow Control Using Credit

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

Report a bug

8.4.2. Credit Allocation Modes

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

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

In both modes, unlimited credit can be allocated for the message count and the total content size.

Report a bug

8.5. Durable Queues

8.5.1. Durable Queues

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

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

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

Report a bug

8.5.2. Persistent Messages

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

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

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

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

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

Report a bug

8.5.3. Create a durable queue in an application

The following example code creates a durable queue called "important-messages":

C++

```
Sender sender = session.createSender("important-messages; {create:always,
node:{durable: True}})
```

Python

```
newqueue = session.sender("important-messages; {create:always, node:{durable:
True})
```

Note that if a queue is declared `durable` *and* `auto-delete`, it is only durable *until* it gets auto-deleted! Carefully consider if this is the behavior that you want.

Report a bug

8.5.4. Mark a message as persistent

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

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

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

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

Table 8.1. Persistent Message and Durable Queue Disk States

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

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

Report a bug

8.5.5. Durable Message State After Restart

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

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

Report a bug

8.5.6. Message Journal

Red Hat Enterprise Messaging allows the size and number of files and caches used for persistence to be configured. There is one journal for each queue; it records each enqueue, dequeue, or transaction event, in order.

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

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

Report a bug

8.5.7. Configure the Message Journal in an application

You can set the file count and file size of the message journal for a queue by specifying `qpid.file_size` and `qpid.file_count` in the `x-declare` arguments of the address used to create a queue:

Python

```
tx = ssn.sender("my-queue;{create: always, node: {durable: True, x-declare:
{arguments: {'qpid.file_size': 20, 'qpid.file_count': 12}}}}")
```

Report a bug

8.6. Transactions

8.6.1. Transactions

Transactional sessions support message transactions - groups of messages whose transmission must succeed or fail atomically. On a transactional session sent messages only become available at the target address on commit. Likewise, received and acknowledged messages are only discarded at their source on commit.

Note that transactions require a reliable link to function.

Report a bug

8.6.2. Transactions Example

The following code demonstrates transactional sessions:

.NET/C#

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

C++

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

Report a bug

Chapter 9. Qpid Management Framework (QMF)

9.1. QMF - Qpid Management Framework

The *Qpid Management Framework* allows the broker to be administered using command messages. 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.

Report a bug

9.2. QMF Versions

Red Hat Enterprise Messaging supports Qpid Management Framework version 2.

QMFv2 offers a number of benefits over QMFv1, including the ability to send QMF messages between nodes in a cluster and across federated links.

For more information on QMFv2, refer to the Apache Qpid QMFv2 Project Page.

QMFv1 calls are possible in Red Hat Enterprise Messaging, but they are not recommended. QMFv1 is deprecated and may be removed in a future release.

Report a bug

9.3. Creating Exchanges from an Application

You can use QMF messages to create exchanges from an application. The following QMF message creates a fanout exchange called `test-fanout`

```
Message(subject='broker', reply_to='qmf.default.topic/direct.6da5bfc3-44fb-4441-
b834-6c5897b9606a;{node:{type:topic}, link:{x-declare:{auto-
delete:True,exclusive:True}}}', correlation_id='1', properties={'qmf.opcode':
'_method_request', 'x-amqp-0-10.app-id': 'qmf2', 'method': 'request'},
content={'_object_id': {'_object_name': 'org.apache.qpid.broker:broker:amqp-
broker'}, '_method_name': 'create', '_arguments': {'strict': True, 'type':
'exchange', 'name': u'test-fanout', 'properties': {'exchange-type': u'fanout'}}})
```

Report a bug

9.4. Broker Exchange and Queue Configuration via QMF

QMF Command messages can be used to create and configure exchanges and queues. The `qpid-config` command-line utility uses QMF messages to perform many of its administration tasks.

Report a bug

9.5. Command Messages

QMF Command Messages are specially formatted map messages sent to the broker's QMF address `qmf.default.direct/broker`.

See Also:

- Chapter 13, *Maps and Lists*

Report a bug

9.6. QMF Command Message Structure

QMF Command Message Content

QMF Command Messages are map messages. A QMF command message contains the keys `_object_id`, `_method_name` and `_arguments`.

The key `_object_id` is mandatory. Its value is a nested map identifying the target of the command. For QMF commands that administer the broker and its resources, the `_object_id` map contains a single value with the 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 of command arguments.

QMF Command Message Properties

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`.

QMF Command Response

To receive a response from the server, set the `reply-to` address of the QMF command message to an address where you can receive messages. After the command message is sent to the broker's QMF address, the response arrives from the `reply-to` address specified. The response message has the `x-amqp-0-10.app-id` property set to `qmf2`.

If the command message is processed as expected, the response message `qmf.opcode` property is set to `_method_response`. If an error was encountered, `qmf.opcode` property will contain the value `_exception`.

The response message content is a map. In the case of a valid response, return values are presented as a nested map against the key `_arguments`. In the case of an exception, details of the exception are within a nested map against the key `_values`.

Report a bug

9.7. Create Command

The QMF `create` command takes five arguments:

type

The type of object to be created, this can be a queue, exchange or binding.

name

The name of the object to be created. 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`.

properties

The specific properties for the object to be created, value is a nested map.

strict

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.

auto_delete_timeout

Optional. If specified upon first declaring an auto-delete queue, specifies a delay, in seconds, after which the deletion will take place. Note: 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 following code example uses QMF to create a queue named `my-queue`. In this example `my-queue` is configured to be auto-deleted after 10 seconds.

Python

```
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()
```

Report a bug

9.8. Delete Command

The QMF `delete` command takes three arguments:

type
The type of object to be created, this can be a queue, exchange or binding.

name
The name of the object to be created. The `name` argument of a queue or exchange is a single value, for example `my-queue`. 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`.

options
A neted map with the key `options`. This is presently unused.

Report a bug

9.9. List Command

The following example QMF message requests a list of exchanges from the broker:

Python

```
Message(subject='broker', reply_to='qmf.default.topic/direct.8b59a7ae-93f1-
4450-9e43-1b0665bf622b;{node:{type:topic}, link:{x-declare:{auto-
delete:True,exclusive:True}}}', correlation_id='1', properties={'qmf.opcode':
'_query_request', 'x-amqp-0-10.app-id': 'qmf2', 'method': 'request'},
content={'_what': 'OBJECT', '_schema_id': {'_class_name': 'exchange'}})
```

The following example QMF message requests a list of queues from the server:

Python

```
Message(subject='broker', reply_to='qmf.default.topic/direct.7f703720-c815-
4c79-986c-354b3963bc76;{node:{type:topic}, link:{x-declare:{auto-
delete:True,exclusive:True}}}', correlation_id='1', properties={'qmf.opcode':
'_query_request', 'x-amqp-0-10.app-id': 'qmf2', 'method': 'request'},
content={'_what': 'OBJECT', '_schema_id': {'_class_name': 'queue'}})
```

Report a bug

9.10. Queue and Exchange Creation using QMF

The following QMF message creates a new queue named `test`:

Python

```
Message(subject='broker', reply_to='qmf.default.topic/direct.8702f596-b112-
427d-b93e-7e0ae28f2ae8;{node:{type:topic}, link:{x-declare:{auto-
delete:True,exclusive:True}}}', correlation_id='1', properties={'qmf.opcode':
'_method_request', 'x-amqp-0-10.app-id': 'qmf2', 'method': 'request'},
content={'_object_id': {'_object_name': 'org.apache.qpid.broker:broker:amqp-
broker'}, '_method_name': 'create', '_arguments': {'strict': True, 'type':
'queue', 'name': u'test', 'properties': {}}})
```

The following QMF message creates a new fanout exchange called `test-fanout`:

Python

```
Message(subject='broker', reply_to='qmf.default.topic/direct.81915d0a-d2e1-
4cf9-9369-921bac725aab;{node:{type:topic}, link:{x-declare:{auto-
delete:True,exclusive:True}}}', correlation_id='1', properties={'qmf.opcode':
'_method_request', 'x-amqp-0-10.app-id': 'qmf2', 'method': 'request'},
content={'_object_id': {'_object_name': 'org.apache.qpid.broker:broker:amqp-
broker'}, '_method_name': 'create', '_arguments': {'strict': True, 'type':
'exchange', 'name': u'test-fanout', 'properties': {'exchange-type': u'fanout'}}})
```

Report a bug

9.11. QMF Events

QMF Events are messages sent to QMF topics to provide notification of broker events. Queue Threshold Alerts are implemented as QMF Events.

The QMF topics are `qmf.default.topic/agent.ind.event.org_apache_qpid_broker.`*`$QMF_Event`*`.#`, where *`$QMF_Event`* is one of the provided QMF Events from the following table:

Table 9.1. QMF Events

QMF Event	Severity	Arguments
clientConnect	inform	rhost, user, properties
clientConnectFail	warn	rhost, user, reason, properties
clientDisconnect	inform	rhost, user, properties
brokerLinkUp	inform	rhost
brokerLinkDown	warn	rhost
queueDeclare	inform	rhost, user, qName, durable, excl, autoDel, altEx, args, disp
queueDelete	inform	rhost, user, qName
exchangeDeclare	inform	rhost, user, exName, exType, altEx, durable, autoDel, args, disp
exchangeDelete	inform	rhost, user, exName
bind	inform	rhost, user, exName, qName, key, args
unbind	inform	rhost, user, exName, qName, key
subscribe	inform	rhost, user, qName, dest, excl, args
unsubscribe	inform	rhost, user, dest
queueThresholdExceeded	warn	qName, msgDepth, byteDepth

See Also:

- Section 8.2.2, “Queue Threshold Alerts”

Report a bug

9.12. QMF Client Connection Events

Changes

- New content for 2.3

Whenever a client connects to or disconnects from the broker, a QMF Event message is generated and sent to a QMF topic.

The QMF topics for these events are:

Table 9.2. QMF Client Connection Event Topics

QMF queue	Purpose
qmf.default.topic/agent.ind.event.org_apache_qpid_broker.clientConnect.#	Client connections
qmf.default.topic/agent.ind.event.org_apache_qpid_broker.clientConnectFail.#	Failed connection attempts
qmf.default.topic/agent.ind.event.org_apache_qpid_broker.clientDisconnect.#	Client disconnections

MRG 2.3 adds additional properties to the QMF Client Connection and Disconnection event messages to match connections and disconnections to specific clients. This enables auditing and troubleshooting. The new properties are:

- `client_ppid` [1]
- `client_pid`
- `client_process`

Here is an example of a QMF client connection event message, demonstrating the client connection information:

```
Fetched Message(
   properties={
      u'qmf.agent': u'apache.org:qpidd:a2ff61bc-19b2-4078-8a7e-9c007151c79c',
      'x-amqp-0-10.routing-key':
u'agent.ind.event.org_apache_qpid_broker.clientConnect.info.apache_org.qpidd.a2ff61
bc-19b2-4078-8a7e-9c007151c79c',
      'x-amqp-0-10.app-id': 'qmf2',
       u'qmf.content': u'_event',
       u'qmf.opcode': u'_data_indication',
       u'method': u'indication'},
       content=[{
          u'_schema_id': {
             u'_package_name': 'org.apache.qpid.broker',
             u'_class_name': 'clientConnect',
             u'_type': '_event',
             u'_hash': UUID('476930ed-01dd-9629-7f84-f42b4b0bc410')},
         u'_timestamp': 1347032560197086881,
         u'_values': {
            u'user': 'anonymous',
            u'properties': {
               u'qpid.session_flow': 1,
               u'qpid.client_ppid': 26139,
               u'qpid.client_pid': 26876,
               u'qpid.client_process': u'spout'},
           u'rhost': '127.0.0.1:5672-127.0.0.1:43276'},
       u'_severity': 6}])

Fri Sep  7 15:42:40 2012 org.apache.qpid.broker:clientConnect user=anonymous
properties={
   u'qpid.session_flow': 1,
   u'qpid.client_ppid': 26139,
   u'qpid.client_pid': 26876,
   u'qpid.client_process': u'spout'}
rhost=127.0.0.1:5672-127.0.0.1:43276
```

Report a bug

9.13. ACL Lookup Query Methods

In MRG 2.3 and above, QMF methods are available to query the ACL Authorization interface.

The Broker must be started with the ACL file that you wish to query, and that ACL file must include sufficient permissions to allow the lookup operations:

```
# Catch 22: allow anonymous to access the lookup debug functions
acl allow-log anonymous create  queue
acl allow-log anonymous all     exchange name=qmf.*
acl allow-log anonymous all     exchange name=amq.direct
acl allow-log anonymous all     exchange name=qpid.management
acl allow-log anonymous access  method   name=Lookup*
```

The QMF methods to query the ACL Authorization interface are `Lookup` and `LookupPublish`.

The `Lookup` method is a general query for any action, object, and set of properties. The `LookupPublish` method is the optimized, per-message fastpath query.

In both methods the result is one of: `allow`, deny, `allow-log`, or deny-`log`.

Method: Lookup

Table 9.3. Method: Lookup

Argument	Type	Direction
userId	long-string	I
action	long-string	I
object	long-string	I
objectName	long-string	I
propertyMap	field-table	I
result	long-string	O

Method: LookupPublish

Table 9.4. Method: LookupPublish

Argument	Type	Direction
userId	long-string	I
exchangeName	long-string	I
routingKey	long-string	I
result	long-string	O

Management Properties and Statistics

The following properties and statistics have been added to reflect command line settings in effect and Acl quota denial activity.

Table 9.5. Broker Management Quota Property

Element	Type	Access	Description
maxConnections	uint16	ReadOnly	Maximum allowed connections

Table 9.6. ACL Management Properties

Element	Type	Access	Description
maxConnectionsPerIp	uint16	ReadOnly	Maximum allowed connections
maxConnectionsPerUser	uint16	ReadOnly	Maximum allowed connections
maxQueuesPerUser	uint16	ReadOnly	Maximum allowed queues
connectionDenyCount	uint64		Number of connections denied
queueQuotaDenyCount	uint64		Number of queue creations denied

Example

Procedure 9.1. ACL Lookup Example

To see a practical example, follow these steps.

1. Start the broker using the example ACL file `acl-test-01-rules.acl` reproduced below, and with `QPID_LOG_ENABLE=debug+:acl`.
2. Run the Python script `acl-test-01.py`.
3. Examine the Python program output and the broker log.

ACL File `acl-test-01-rules.acl`

```
# acl-test-rules-00.acl
# 27-march-2012

group admins moe@COMPANY.COM \
            larry@COMPANY.COM \
      curly@COMPANY.COM \
      shemp@COMPANY.COM

group auditors aaudit@COMPANY.COM baudit@COMPANY.COM caudit@COMPANY.COM \
               daudit@COMPANY.COM eaduit@COMPANY.COM eaudit@COMPANY.COM

group tatunghosts tatung01@COMPANY.COM \
                  tatung02/x86.build.company.com@COMPANY.COM \
                  tatung03/x86.build.company.com@COMPANY.COM \
                  tatung04/x86.build.company.com@COMPANY.COM \
                  HTTP/tatung-test1.eng.company.com@COMPANY.COM

group publishusers publish@COMPANY.COM x-pubs@COMPANY.COM

# Admins: This should be the *only* group which ever gets "all" access
# to anything. Everything/everyone else must not be as permissive
acl allow-log admins all all

# Catch 22: allow anonymous to access the lookup debug functions
acl allow-log anonymous create  queue
acl allow-log anonymous all     exchange name=qmf.*
acl allow-log anonymous all     exchange name=amq.direct
acl allow-log anonymous all     exchange name=qpid.management
acl allow-log anonymous access  method   name=Lookup*

acl allow all publish exchange name=''

# Auditors
acl allow-log auditors all exchange name=company.topic routingkey=private.audit.*

# Tatung
acl allow-log tatunghosts  publish exchange name=company.topic  routingkey=tatung.*
acl allow-log tatunghosts  publish exchange name=company.direct routingkey=tatung-
service-queue

# Publish
acl allow-log publishusers create queue
acl allow-log publishusers publish exchange name=qpid.management routingkey=broker
acl allow-log publishusers publish exchange name=qmf.default.topic routingkey=*
acl allow-log publishusers publish exchange name=qmf.default.direct routingkey=*

# Consumers - everyone
acl allow-log all bind exchange name=company.topic  routingkey=tatung.*
acl allow-log all bind exchange name=company.direct routingkey=tatung-service-
queue

acl allow-log all consume queue

acl allow-log all access exchange
acl allow-log all access queue

acl allow-log all create queue name=tmp.* durable=false autodelete=true
exclusive=true policytype=ring

# All else is denied
acl deny-log all all
```

Python Script `acl-test-01.py`

```
# acl-test-00.py
# test driver for QPID-3918 lookup hooks.
#
# The broker is to use acl-test-00-rules.acl.
#
import sys
import qpid
import qmf

totalLookups = 0
failLookups  = 0
exitOnError  = True

#
# Run a type 1 lookup
# This is the general lookup
#
def Lookup(acl, userName, action, aclObj, aclObjName, propMap, expectedResult =
''):
    global totalLookups
    global failLookups
    totalLookups += 1
    result = acl.Lookup(userName, action, aclObj, aclObjName, propMap)
    suffix = ''
    if (expectedResult != ''):
        if (result.result != expectedResult):
            failLookups += 1
            suffix = ', [ERROR: Expected ' + expectedResult + "]"
            if (result.result is None):
                suffix = suffix + ', [' + result.text + ']'
    print 'Lookup : [name:', userName, ", action: ", action, ", object: ", aclObj,
\
        ", objName: '", aclObjName, "', properties: ", propMap, \
        "], [Result: ", result.result, "]", suffix
    if (exitOnError and failLookups > 0):
        sys.exit()

#
# Run a type 2 lookup
# This is a specific PUBLISH EXCHANGE ['user', 'exchangeName', 'routingKey']
lookup
#
def LookupPublish(acl, userName, exchName, keyName, expectedResult = ''):
    global totalLookups
    global failLookups
    totalLookups += 1
    result = acl.LookupPublish(userName, exchName, keyName)
    suffix = ''
    if (expectedResult != ''):
        if (result.result != expectedResult):
            failLookups += 1
            suffix = ', [ERROR: Expected ' + expectedResult + "]"
            if (result.result is None):
                suffix = suffix + ', [' + result.text + ']'
    print 'LookupPublish : [name:', userName, \
        ", exchName: '", exchName, "', key: ", keyName, \
        "], [Result: ", result.result, "]", suffix
    if (exitOnError and failLookups > 0):
        sys.exit()

#
# AllBut
#
# Given All names and some names we don't want,
# return the All list with the targets removed
#
def AllBut(allList, removeList):
    tmpList = allList[:]
    for item in removeList:
        try:
            tmpList.remove(item)
        except Exception, e:
            print "ERROR in AllBut() \nallList =  %s \nremoveList =  %s \nerror =
%s " \
                % (allList, removeList, e)
    return tmpList


#
# Main
#
# Fire up a session and get the acl methods
#

from qmf.console import Session
sess = Session()
broker = sess.addBroker()
acls = sess.getObjects(_class="acl", _package="org.apache.qpid.acl")
acl = acls[0]
# print acl.getMethods() # just to see the method names available

#
# define some group lists
#
g_admins = ['moe@COMPANY.COM', \
           'larry@COMPANY.COM', \
     'curly@COMPANY.COM', \
     'shemp@COMPANY.COM']

g_auditors = [ 'aaudit@COMPANY.COM','baudit@COMPANY.COM','caudit@COMPANY.COM', \
               'daudit@COMPANY.COM','eaduit@COMPANY.COM','eaudit@COMPANY.COM']

g_tatunghosts = ['tatung01@COMPANY.COM', \
                 'tatung02/x86.build.company.com@COMPANY.COM', \
                 'tatung03/x86.build.company.com@COMPANY.COM', \
                 'tatung04/x86.build.company.com@COMPANY.COM', \
                 'HTTP/tatung-test1.eng.company.com@COMPANY.COM']

g_publishusers = ['publish@COMPANY.COM', 'x-pubs@COMPANY.COM']

g_public = ['jpublic@COMPANY.COM', 'me@yahoo.com']

g_all = g_admins + g_auditors + g_tatunghosts + g_publishusers + g_public

action_all =
['consume','publish','create','access','bind','unbind','delete','purge','update']

#
# Run some tests
#
print '#'
print '# admin'
print '#'

for u in g_admins:
    Lookup(acl, u, "create", "queue", "anything", {"durable":"true"}, "allow-log")

print '#'
print '# auditors'
print '#'

uInTest = g_auditors + g_admins
uOutTest = AllBut(g_all, uInTest)

for u in uInTest:
    LookupPublish(acl, u, "company.topic", "private.audit.This", "allow-log")

for u in uInTest:
    for a in action_all:
        Lookup(acl, u, a, "exchange", "company.topic",
{"routingkey":"private.audit.This"}, "allow-log")

for u in uOutTest:
    LookupPublish(acl, u, "company.topic", "private.audit.This", "deny-log")
    Lookup(acl, u, "bind", "exchange", "company.topic",
{"routingkey":"private.audit.This"}, "deny-log")

print '#'
print '# tatungs'
print '#'

uInTest = g_admins + g_tatunghosts
uOutTest = AllBut(g_all, uInTest)

for u in uInTest:
    LookupPublish(acl, u, "company.topic",  "tatung.this2",         "allow-log")
    LookupPublish(acl, u, "company.direct", "tatung-service-queue", "allow-log")

for u in uOutTest:
    LookupPublish(acl, u, "company.topic",  "tatung.this2",         "deny-log")
    LookupPublish(acl, u, "company.direct", "tatung-service-queue", "deny-log")

for u in uOutTest:
    for a in ["bind", "access"]:
        Lookup(acl, u, a, "exchange", "company.topic",
{"routingkey":"tatung.this2"},          "allow-log")
        Lookup(acl, u, a, "exchange", "company.direct", {"routingkey":"tatung-
service-queue"}, "allow-log")

print '#'
print '# publishusers'
print '#'

uInTest = g_admins + g_publishusers
uOutTest = AllBut(g_all, uInTest)

for u in uInTest:
    LookupPublish(acl, u, "qpid.management",    "broker",   "allow-log")
    LookupPublish(acl, u, "qmf.default.topic",  "this3",    "allow-log")
    LookupPublish(acl, u, "qmf.default.direct", "this4",    "allow-log")

for u in uOutTest:
    LookupPublish(acl, u, "qpid.management",    "broker",   "deny-log")
    LookupPublish(acl, u, "qmf.default.topic",  "this3",    "deny-log")
    LookupPublish(acl, u, "qmf.default.direct", "this4",    "deny-log")

for u in uOutTest:
    for a in ["bind"]:
        Lookup(acl, u, a, "exchange", "qpid.management",
{"routingkey":"broker"}, "deny-log")
        Lookup(acl, u, a, "exchange", "qmf.default.topic",
{"routingkey":"this3"},  "deny-log")
```

```
        Lookup(acl, u, a, "exchange", "qmf.default.direct",
{"routingkey":"this4"},  "deny-log")
    for a in ["access"]:
        Lookup(acl, u, a, "exchange", "qpid.management",
{"routingkey":"broker"}, "allow-log")
        Lookup(acl, u, a, "exchange", "qmf.default.topic",
{"routingkey":"this3"},  "allow-log")
        Lookup(acl, u, a, "exchange", "qmf.default.direct",
{"routingkey":"this4"},  "allow-log")

#
# Report statistics
#
print 'Total  Lookups: ', totalLookups
print 'Failed Lookups: ', failLookups

#
# Close the session
#
sess.close()
```

Report a bug

9.14. Using QMF in a Cluster

To use QMF messages in a cluster, use QMF version 2. QMF version 1 messages cannot be used in a cluster.

Report a bug

[1] Not available in the Java client

Chapter 10. The Qpid Messaging API

10.1. Handling Exceptions

10.1.1. Messaging Exceptions Reference

In the asynchronous and decoupled environment of a messaging application, exceptions are thrown for both local error conditions and error conditions or failures that occur remotely. Developing a robust application requires that you anticipate and handle a wide range of possible exceptions, some of which are not immediately obvious from the context of the method itself.

Report a bug

10.1.2. C++ Messaging Exceptions Class Hierarchy

The following are the exceptions thrown by the C++ API, and the circumstances under which they are thrown. The source code for the exceptions can be viewed in the Apache Qpid svn repository.

MessagingException
The base class for Messaging exceptions.

InvalidOptionString : public MessagingException
Thrown when the syntax of the option string used to configure a connection is not valid.

KeyError : public MessagingException
Thrown to indicate a failed lookup of some local object. For example when attempting to retrieve a session, sender or receiver by name.

LinkError : public MessagingException
Base class for exceptions thrown to indicate a failed lookup of some local object.

AddressError : public LinkError
Thrown to indicate a failed lookup of some local object. For example when attempting to retrieve a session, sender or receiver by name.

ResolutionError : public AddressError
Thrown when a syntactically correct address cannot be resolved or used.

AssertionFailed : public ResolutionError
Thrown when creating a sender or receiver for an address for which some asserted property of the node is not matched.

NotFound : public ResolutionError
Thrown on attempts to create a sender or receiver to a non-existent node.

MalformedAddress : public AddressError
Thrown when an address string with invalid syntax is used.

ReceiverError : public LinkError

FetchError : public ReceiverError

NoMessageAvailable : public FetchError
Thrown by Receiver::fetch(), Receiver::get() and Session::nextReceiver() to indicate that there no message was available before the timeout specified.

SenderError : public LinkError

SendError : public SenderError

TargetCapacityExceeded : public SendError
Thrown to indicate that the sender attempted to send a message that would result in the target node on the peer exceeding a preconfigured capacity.

SessionError : public MessagingException

TransactionError : public SessionError

TransactionAborted : public TransactionError
Thrown on Session::commit() if reconnection results in the transaction being automatically aborted.

UnauthorizedAccess : public SessionError
Thrown to indicate that the application attempted to do something for which it was not authorised by its peer.

UnauthorizedAccess : public SessionError

ConnectionError : public MessagingException

TransportFailure : public MessagingException
Thrown to indicate loss of underlying connection. When auto-reconnect is used this will be caught by the library and used to trigger reconnection attempts. If reconnection fails (according to whatever settings have been configured), then an instance of this class will be thrown to signal that.

Report a bug

10.1.3. Connection Exceptions

Note: Unless fully qualified, all exceptions listed are in the `qpid::messaging` namespace.

Connection::Connection(const std::string&, const qpid::types::Variant::Map&)

`MessagingException` if any of the options in the supplied map are not recognised.

`qpid::types::InvalidConversion` if any of the option values are of the wrong type.

Connection::Connection(const std::string& url, const std::string& options)

`MessagingException` if any of the options in the supplied map are not recognised.

`qpid::types::InvalidConversion` if any of the option values are of the wrong type.

`InvalidOptionString` if the format of the option string is invalid.

Connection::setOption(const std::string& name, const qpid::types::Variant& value)

`MessagingException` if the named option is not recognised.

`qpid::types::InvalidConversion` if the option value is of the wrong type.

Connection::open()

`qpid::Url::Invalid` if the url is not valid (this may be the url supplied on construction or any of the reconnect_urls supplied via options).

`TransportFailure` if a connection could not be established.

`ConnectionError` for any other failure, including where the broker sends a connection.close control before the AMQP 0-10 defined connection handshake completes.

`qpid::types::InvalidConversion` if the broker sends an improperly encoded value for the '`known-host`' field of the `connection.open-ok control` as defined by AMQP 0-10 specification.

Connection::isOpen()

Does not throw exceptions.

Connection::close()

`TargetCapacityExceeded` if any of the sessions established for the connection have attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if any of the sessions established for the connection have attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection just before the client does).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session while the close is in progress).

`TransportFailure` if a connection was lost while trying to perform the close 'handshake' with the broker.

Connection::createTransactionalSession(const std::string& name)

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected which could happen on enabling transactions for the session (e.g. if the broker in question did not support transactions).

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of the session before it becomes active).

`TransportFailure` if the connection was lost (and if automatic reconnect is enabled could not be re-established).

`qpid::Url::Invalid` if reconnect is enabled and a url in the `reconnect_urls` option list is invalid.

`qpid::types::InvalidConversion` if the broker were to send an improperly encoded value for the '`known-host`' field of the `connection.open-ok` control as defined by AMQP 0-10 specification.

Connection::createSession(const std::string&)

`ConnectionError` if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of the session before it becomes active).

`TransportFailure` if the connection was lost (and if automatic reconnect is enabled could not be re-established).

`qpid::Url::Invalid` if reconnect is enabled and a url in the `reconnect_urls` option list is invalid.

`qpid::types::InvalidConversion` if the broker were to send an improperly encoded value for the '`known-host`' field of the connection.open-ok control as defined by AMQP 0-10 specification.

Connection::getSession(const std::string&)

`KeyError` if no session for the specified name exists.

Connection::getAuthenticatedUsername()

Does not throw any exception.

Report a bug

10.1.4. Session Exceptions

Note: Unless fully qualified, all exceptions listed are in the `qpid::messaging` namespace.

Session::close()

TargetCapacityExceeded if the session has attempted to send a message that would result in a queue exceeding configured limits.

UnauthorizedAccess if the session has attempted to perform an operation for which it has not been granted permission.

SessionError if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

ConnectionError if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection).

MessagingException if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

TransportFailure if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::commit()

TransactionAborted if the original AMQP 0-10 session is lost, e.g. due to failover, forcing an automatic rollback.

TargetCapacityExceeded if the session has attempted to send a message that would result in a queue exceeding configured limits.

UnauthorizedAccess if the session has attempted to perform an operation for which it has not been granted permission.

SessionError if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

ConnectionError if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection).

MessagingException if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

TransportFailure if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::rollback()

TargetCapacityExceeded if the session has attempted to send a message that would result in a queue exceeding configured limits.

UnauthorizedAccess if the session has attempted to perform an operation for which it has not been granted permission.

SessionError if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

ConnectionError if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection).

MessagingException if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

TransportFailure if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::acknowledge(bool)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::acknowledge(Message&, bool)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::acknowledgeUpTo(Message&, bool)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::reject(Message&)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

Throws `SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::release(Message&)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

TransportFailure if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::sync(bool)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::getReceivable()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::getUnsettledAcks()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

TransportFailure if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::nextReceiver(Receiver&, Duration)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::nextReceiver(Duration)

`Receiver::NoMessageAvailable` if no message became available in time.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

Throws `SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::createSender(const Address&)

`ResolutionError` if there is an error in resolving the address.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::createSender(const std::string&)

`ResolutionError` if there is an error in resolving the address.

`MalformedAddress` if the syntax of the address string is not valid.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::createReceiver(const Address&)

`ResolutionError` if there is an error in resolving the address.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a connection.close control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::createReceiver(const std::string&)

`ResolutionError` if there is an error in resolving the address.

`MalformedAddress` if the syntax of the address string is not valid.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Session::getSender(const std::string&)

`KeyError` if there is no sender for the specified name.

Session::getReceiver(const std::string&)

KeyError if there is no receiver for the specified name.

Session::checkError()

`qpid::messaging::SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`qpid::messaging::ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`qpid::messaging::MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

Session::getConnection()

Does not throw exceptions.

Session::hasError()

Does not throw exceptions.

Report a bug

10.1.5. Sender Exceptions

Note: Unless fully qualified, all exceptions listed are in the `qpid::messaging` namespace.

Sender::send(const Message& message, bool)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a session.detached control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Sender::close()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Sender::setCapacity(uint32_t)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Sender::getUnsettled()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Sender::getAvailable()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Sender::getCapacity()
Does not throw exceptions.

Sender::getName()
Does not throw exceptions.

Sender::getSession()
Does not throw exceptions.

Report a bug

10.1.6. Receiver Exceptions

Note: Unless fully qualified, all exceptions listed are in the `qpid::messaging` namespace.

Receiver::get(Message& message, Duration timeout=Duration::FOREVER)
`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::Message get(Duration timeout=Duration::FOREVER)
`NoMessageAvailable` if there is no message to give after waiting for the specified timeout, or if the Receiver is closed, in which case `isClose()` will be true.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::fetch(Message& message, Duration timeout=Duration::FOREVER)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an execution.exception command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::fetch(Duration timeout=Duration::FOREVER)

`NoMessageAvailable` if there is no message to give after waiting for the specified timeout, or if the Receiver is closed, in which case `isClose()` will be true.

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::setCapacity(uint32_t)

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::getAvailable()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::getUnsettled()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::close()

`TargetCapacityExceeded` if the session has attempted to send a message that would result in a queue exceeding configured limits.

`UnauthorizedAccess` if the session has attempted to perform an operation for which it has not been granted permission.

`SessionError` if an `execution.exception` command, as defined in AMQP 0-10, is received from the broker to which the client is connected.

`ConnectionError` if the broker to which the client is connected sends a `connection.close` control (i.e. if broker initiates closing of an active connection).

`MessagingException` if the broker to which the client is connected sends a `session.detached` control (i.e. if broker initiates closing of an active session).

`TransportFailure` if a connection was lost (and if automatic reconnect is enabled could not be re-established).

Receiver::isClosed()
Does not throw exceptions.

Receiver::getCapacity()
Does not throw exceptions.

Receiver::getName()
Does not throw exceptions.

Receiver::getSession()
Does not throw exceptions.

Report a bug

Chapter 11. Addresses

11.1. x-declare Parameters

The following parameters may be supplied in the `x-declare` part of an address string:

Table 11.1.

Parameter	Usage
`auto-delete`	boolean specifying if the queue/exchange should be auto-deleted
`exclusive`	boolean specifying exclusiveness of the queue/exchange
`alternate-exchange`	alternate exchange where messages shall be routed to when this queue is deleted / the exchange fails to find a matching bind for a message
`arguments`	a nested map with arguments available specifically for the queue / exchange. Refer to https://cwiki.apache.org/confluence/display/qpid/Qpid+extensions+to+AMQP for further details.

Report a bug

11.2. Connection Options

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

Report a bug

11.3. Setting Connection Options

There are two different ways to set connection options. The first is to do it in the Connection constructor:

Python

```
connection = Connection("localhost:5672", reconnect = True, reconnect_urls =
"amqp:tcp:127.0.0.1:5674", heartbeat = 1)
try:
  connection.open()
```

C++

```
Connection connection("localhost:5672", "{reconnect: true,
reconnect_urls:'amqp:tcp:127.0.0.1:5674', reconnect:true, heartbeat: 1}");
try {
    connection.open();
```

.NET/C#

```
Connection connection= new Connection("localhost:5672", "{reconnect: true,
reconnect_urls:'amqp:tcp:127.0.0.1:5674', reconnect:true, heartbeat: 1}");
try {
    connection.Open();
```

The second approach is to do it through the Connection properties:

Python

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

C++

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

.NET/C#

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

Report a bug

11.4. Connection Options Reference

Table 11.2. 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_urls`	Broker address list	A list of one or more brokers to attempt communication with when a connection fails.
`reconnect_timeout`	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. Heartbeats should be defined to, at most, 1/2 of TCP retransmission overall-time. By default, TCP retransmission time is around 15 minutes on Linux and 12

		seconds on Windows.
`protocol`	string	Sets the underlying protocol used. The default option is *`tcp`*. To enable ssl, set to *`ssl`*. The C++ client additionally supports *`rdma`*.
`tcp_nodelay`	boolean	Set *`tcp_no_delay`*, i.e. disable Nagle algorithm.

Report a bug

Chapter 12. Message Timestamping

12.1. Message Timestamping

Messages can be timestamped with the date and time of their arrival at the broker. By default timestamping of messages is turned off.

Report a bug

12.2. Enable Message Timestamping at Broker Start-up

To enable message timestamping at broker start-up, start the broker with the `--enable-timestamp yes` argument:

```
./qpidd --enable-timestamp yes
```

Report a bug

12.3. Enable Message Timestamping from an Application

QMF command messages can be used to enable and disable timestamping from an application, with no need to restart the broker.

The QMF methods `getTimestampConfig` and `setTimestampConfig` get and set the timestamping configuration.

getTimestampConfig
Returns `True` if received messages are timestamped.

setTimestampConfig
Set `True` to enable timestamping received messages, `False` to disable timestamping.

Report a bug

12.4. Access a Message Timestamp in Python

The following code fragment checks for and extracts the message timestamp from a received message.

```
try:
    msg = receiver.fetch(timeout=1)
    if "x-amqp-0-10.timestamp" in msg.properties:
        print("Timestamp=%s" % str(msg.properties["x-amqp-0-10.timestamp"]))
except Empty:
    pass
```

Report a bug

12.5. Access a Message Timestamp in C++

The following code fragment checks for and extracts the message timestamp from a received message.

```
messaging::Message msg;
if (receiver.fetch(msg, messaging::Duration::SECOND*1)) {
    if (msg.getProperties().find("x-amqp-0-10.timestamp") !=
msg.getProperties().end()) {
        std::cout << "Timestamp=" <<
msg.getProperties()["x-amqp-0-10.timestamp"].asString() << std::endl;
    }
}
```

Report a bug

12.6. Using AMQ 0-10 Message Property Keys for Timestamping

If the timestamp delivery property is set in an incoming message (*delivery-properties.timestamp*), the timestamp value can be accessed using the *x-amqp-0-10.timestamp* message property.

See Also:

- Chapter 19, *The AMQP 0-10 mapping*

Report a bug

Chapter 13. Maps and Lists

13.1. Maps and Lists in Message Content

Messaging applications frequently need to exchange data across languages and platforms. Messages can contain maps and lists.

Report a bug

13.2. Map and List Representation in Native Data Types

Table 13.1. Map and List Representation in Supported Languages

Language	map	list
Python	`dict`	`list`
C++	`Variant::Map`	`Variant::List`
Java	`MapMessage`	`ListMessage` [a]
.NET	`Dictionary<string, object>`	`Collection<object>`

[a] MRG 2.3+

Report a bug

13.3. Differences between Qpid and JMS Map Message Content

In versions of Red Hat Enterprise Message (MRG) up 2.2, the Qpid JMS client supports `MapMessages` whose values can be nested maps or lists. This is not standard JMS behavior.

MRG 2.3 adds the `ListMessage` type for lists.

See Also:

- Section 20.6, “JMS ListMessage”

Report a bug

13.4. 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:

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

Report a bug

13.5. Python Data Types in Maps

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 13.2. 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

Report a bug

13.6. 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:

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);
```

Report a bug

13.7. C++ Data Types in Maps

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 13.3. 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

Report a bug

13.8. 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:

.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;
        }
    }
```

```
}
```

Report a bug

13.9. C# Data Types and .NET bindings

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

Table 13.4. 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
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.

Report a bug

Chapter 14. The Request/Response Pattern

14.1. The Request/Response Pattern

Request/Response applications use the `reply-to` message property 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.

Report a bug

14.2. Request/Response C++ Example

This example is 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.

C++

```
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.

C++

```
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.

Report a bug

Chapter 15. Performance Tips

15.1. Apache Qpid Programming for Performance

- Consider prefetching messages for receivers. 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. Rather than acknowledging each message individually, consider issuing acknowledgments after n messages and/or after a particular duration has elapsed.
- Tune the sender capacity. 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.

Report a bug

Chapter 16. Cluster Failover

16.1. Messaging Clusters

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

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

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

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

Note

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

Report a bug

16.2. Cluster Failover in C++

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. 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:

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

Report a bug

16.3. Cluster Failover in Python

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. A client can use this information to dynamically keep the list of reconnection URLs up to date. The following example configures cluster failover in Python:

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

Report a bug

16.4. Cluster Failover in C#

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. A client can use this information to dynamically keep the list of reconnection URLs up to date. The following example configures cluster failover in C#:

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

Report a bug

16.5. Failover Behavior in Java JMS Clients

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

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

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

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

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

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

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

For example:

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

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

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

The failover property can take three values:

Failover Modes

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

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

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

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

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

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

Report a bug

Chapter 17. Logging

17.1. Logging in C++

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

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

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

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

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

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

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

Report a bug

17.2. Logging in Python

The Python client library supports logging using the standard Python logging module.

The `basicConfig()` logging method reports all warnings and errors:

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

The `qpidd` daemon alllows you 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, run `$ pydoc qpid.log`.

Report a bug

17.3. Change the logging level at runtime

The logging level of the broker can be changed at runtime, without restarting. This is useful to increase the level of logging detail while debugging, then return it to a lower level.

The Qpid Management Framework Broker object has a `setLogLevel` method to control the logging level. The following C++ code demonstrates calling this method to set the logging level.

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

#include <iostream>

using namespace std;
using namespace qpid::messaging;
using namespace qpid::types;

int main(int argc, char** argv) {
  if (argc < 2) {
    cerr << "Invalid number of parameters, expecting log level (info, trace,
warning or so)" << endl;
    return 1;
  }
  string log_level = argv[1];

  Connection connection(argc>2?argv[2]:"localhost:5672");
  connection.open();
  Session session = connection.createSession();
  Sender sender = session.createSender("qmf.default.direct/broker");
  Address responseQueue("#reply-queue; {create:always, node:{x-declare:{auto-
delete:true}}}");
  Receiver receiver = session.createReceiver(responseQueue);

  Message message;
  Variant::Map content;
  Variant::Map OID;
  Variant::Map arguments;

  OID["_object_name"] = "org.apache.qpid.broker:broker:amqp-broker";
  arguments["level"] = log_level;

  content["_object_id"] = OID;
  content["_method_name"] = "setLogLevel";
  content["_arguments"] = arguments;

  encode(content, message);
  message.setReplyTo(responseQueue);
  message.setProperty("x-amqp-0-10.app-id", "qmf2");
  message.setProperty("qmf.opcode", "_method_request");
  message.setContentType("amqp/map");

  sender.send(message, true);

  /* receive a response from the broker & check our request was successfully
processed */
  Message response;
  if (receiver.fetch(response,qpid::messaging::Duration(30000)) == true) {
    qpid::types::Variant::Map recv_props = response.getProperties();
    if (recv_props["x-amqp-0-10.app-id"] == "qmf2")
      if (recv_props["qmf.opcode"] == "_method_response")
        std::cout << "Response: OK" << std::endl;
      else if (recv_props["qmf.opcode"] == "_exception")
        std::cerr << "Error: " << response.getContent() << std::endl;
      else
        std::cerr << "Invalid response received!" << std::endl;
    else
      std::cerr << "Invalid response not of qmf2 type received!" << std::endl;
  }
  else
    std::cout << "Timeout: No response received within 30 seconds!" << std::endl;

  receiver.close();
  sender.close();
  session.close();
  connection.close();
  return 0;
      }
```

1. Save the example code to a file `set_log_level.cpp`.
2. Modify the Connection URL in the code to resolve to your broker. At the moment it is set to connect to a broker running on port 5672 on the local machine.
3. Compile the example code:

   ```
   g++ -Wall -lqpidclient -lqpidcommon -lqpidmessaging -lqpidtypes -o
   set_log_level set_log_level.cpp
   ```

4. Use the complied program to change the log level of the broker:

   ```
   ./set_log_level "trace+"
   ```

5. To observe the change in the logging level, tail the server log as you run the program.

Chapter 18. Security

18.1. Security features provided by Qpid

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

Report a bug

18.2. Authentication

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

Report a bug

18.3. SASL Support in Windows Clients

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

This is likely to change in a future release.

Report a bug

18.4. Enable Kerberos authentication

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");
```

Report a bug

18.5. Enable SSL

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

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

Report a bug

18.6. SSL Client Environment Variables for C++ Clients

Table 18.1. 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.

Report a bug

Chapter 19. The AMQP 0-10 mapping

19.1. The AMQP 0-10 mapping

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, 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 19.1. Mapping to AMQP 0-10 Message Properties

Python API	C++ API [a]	AMQP 0-10 Property [b]
`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` [c]
`msg.correlation_id`	`msg.{get,set}CorrelationId()`	`mp.correlation_id`
`msg.durable`	`msg.{get,set}Durable()`	`dp.delivery_mode == delivery_mode.persistent` [d]
`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`

[a] The .NET Binding for C++ Messaging provides all the message and delivery properties described in the C++ API.
[b] In these entries, *mp* refers to an AMQP message property, and *dp* refers to an AMQP delivery property.
[c] The reply_to is converted from the protocol representation into an address.
[d] Note that msg.durable is a boolean, not an enum.

Report a bug

19.2. AMQ 0-10 Message Property Keys

The Qpid Messaging API recognizes special message property keys and automatically provides a mapping to their corresponding AMQP 0-10 definitions.

For example, when sending a message, if the properties contain an entry for *`x-amqp-0-10.app-id`*, its value will be used to set the *`message-properties.app-id`* property in the outgoing message. Likewise, if an incoming message has *`message-properties.app-id`* set, its value can be accessed via the *`x-amqp-0-10.app-id`* message property key.

Similarly, when sending a message, if the properties contain an entry for *`x-amqp-0-10.content-encoding`*, its value will be used to set the *`message-properties.content-encoding`* property in the outgoing message. Likewise, if an incoming message has *`message-properties.content-encoding`* set, its value can be accessed via the *`x-amqp-0-10.content-encoding`* message property key.

The routing key (*`delivery-properties.routing-key`*) in an incoming messages can be accessed via the *`x-amqp-0-10.routing-key`* message property.

Report a bug

19.3. AMQP Routing Key and Message Subject

Whenever you send a message using the Qpid Messaging API in Red Hat Enterprise Messaging, the `x-amqp-0-10.routing-key` property is set to the value of the message subject, with one exception.

Any message that has a subject explicitly set has its subject preserved and the AMQP routing key set to the message subject when it is sent.

When a message has no subject manually set, its subject is set by the sender, if the sender's destination address contains a subject.

Take for example, the following sender:

```
sender = session.sender('amq.topic/SubjectX')
```

Given these two messages:

```
msg1 = Message('A message with no subject')

msg2 = Message('A message with a subject')
msg2.subject = 'SubjectY'
```

`msg1` has its subject and AMQP routing key set to '`SubjectX`'. `msg2` retains its subject '`SubjectY`', and has its AMQP routing key set to '`SubjectY`'.

There are only two other cases.

The first is when a message with no subject is sent via a sender with no subject in its destination address. For example, in Python:

```
sender = session('amq.topic')
msg = Message('No subject, and none assigned by the sender')
sender.send(msg)
```

In this case the message is sent with a blank subject and a blank AMQP routing key.

The second, and only exceptional case, is when a message with a blank subject and a manually assigned AMQP routing key is sent via a sender with no subject in its destination address. For example, in Python:

```
sender = session('amq.topic')
msg = Message('No subject, but a manually assigned AMQP routing key')
msg.properties['x-amqp-0-10.routing-key'] = 'amqp-SubjectX'
sender.send(msg)
```

In this case, the message is sent with a blank subject, and the arbitrary AMQP routing key assigned.

Note that in this case the message will not route in a Red Hat Enterprise Messaging topic exchange. The `amqp-0-10.routing-key` may be useful in an interoperability scenario, but in Red Hat Enterprise Messaging the message `subject` is used for routing.

The following Python program demonstrates the various permutations of interaction between message subject, sender destination address subject, and message routing key:

```
import sys
from qpid.messaging import *

# This program demonstrates that the x-amqp-0-10.routing-key
# (1) is (re)set to the message subject when the message has a subject or
#   is sent via a sender that has a subject
# (2) is not a valid basis for routing in a topic exchange
#   -  the topic exchange will not route a message to a queue

def sendmsg(msg, note = ''):
  global rxplain, rxsubject, txplain, txsubject, ssn, testcount

  msg.properties['sender'] = 'Plain Sender'
  txplain.send(msg)

  msg.properties['sender'] = 'SubjectX Sender'
  txsubject.send(msg)

  if testcount > 0:
    x = raw_input('\nPress Enter for the next test message')
    print '\n==================================================\n'

  testcount = testcount + 1
  print '\nScenario ' + str(testcount)
  print '\nSent message:\n'
  subject = 'Blank'
  if msg.subject:
    subject = msg.subject
  print 'Subject:\t' + subject
  routekey = 'Blank'
  if 'x-amqp-0-10.routing-key' in msg.properties:
    routekey = msg.properties['x-amqp-0-10.routing-key']
  print 'Routing Key:\t' + routekey

  msgcount = 0

  print '\nThe queue listening for all messages received:'
  try:
    while True:
      rxmsg = rxplain.fetch(timeout = 1)
      subject ='Blank'
      if rxmsg.subject:
        subject = rxmsg.subject
      routekey = 'Blank'
      if 'x-amqp-0-10.routing-key' in rxmsg.properties:
        routekey = rxmsg.properties['x-amqp-0-10.routing-key']
      print '\nSubject:\t' + subject
      print 'Routing Key:\t' + routekey
      print 'Sent via:\t' + rxmsg.properties['sender']
      msgcount = 1
      ssn.acknowledge(rxmsg)
  except:
    pass

  if msgcount == 0:
    print 'Nothing\n'
  else:
    msgcount = 0

  print '\nThe queue listening for SubjectX messages received:'
  try:
    while True:
      rxmsg = rxsubject.fetch(timeout = 1)
      subject ='Blank'
      if rxmsg.subject:
        subject = rxmsg.subject
      routekey = 'Blank'
      if 'x-amqp-0-10.routing-key' in rxmsg.properties:
        routekey = rxmsg.properties['x-amqp-0-10.routing-key']
      print '\nSubject:\t' + subject
      print 'Routing Key:\t' + routekey
      print 'Sent via:\t' + rxmsg.properties['sender']
      msgcount = 1
      ssn.acknowledge(rxmsg)
  except:
    pass

  if msgcount == 0:
    print 'Nothing\n'

  if note != '':
    print '\nNote: ' + note + "\n"

connection = Connection("localhost:5672")
connection.open()

try:
  ssn = connection.session()

  # we create our receivers here so that queues are created to hold the messages
sent
  rxplain = ssn.receiver("amq.topic")
  rxsubject = ssn.receiver("amq.topic/SubjectX")

  txplain = ssn.sender("amq.topic")
  txsubject = ssn.sender("amq.topic/SubjectX")

  testcount = 0

  msg = Message("Plain message, no subject")
  sendmsg(msg, "a subject sender writes the subject and routing key when a message
has no subject, a plain sender does not")

  msg = Message("Message with subject")
  msg.subject = "SubjectX"
  sendmsg(msg, "a plain sender writes the routing key if the message has a
subject")

  msg = Message("Message with a different subject")
  msg.subject = "SubjectY"
  sendmsg(msg, "a subject sender does not rewrite a subject, both senders use the
message subject to write routing key")

  msg = Message("Message with routing key")
  msg.properties["x-amqp-0-10.routing-key"] = "SubjectX"
  sendmsg(msg, "a routing key is not sufficient to route to a queue - the match is
on subject")

  msg = Message("Message with different routing key")
  msg.properties["x-amqp-0-10.routing-key"] = "SubjectY"
  sendmsg(msg, "the only case where you can manually set a non-blank routing key is
a message with a blank subject, sent via a plain sender")

  msg = Message("Message with different routing key and subject")
  msg.properties["x-amqp-0-10.routing-key"] = "SubjectY"
  msg.subject = "SubjectZ"
  sendmsg(msg, "all messages with subjects and all messages sent via a subject
sender have their routing key rewritten")
```

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

Report a bug

19.4. Using AMQ 0-10 Message Property Keys for Timestamping

If the timestamp delivery property is set in an incoming message (*`delivery-properties.timestamp`*), the timestamp value can be accessed using the *`x-amqp-0-10.timestamp`* message property.

See Also:

- Chapter 12, *Message Timestamping*

Report a bug

Chapter 20. Using the Qpid JMS client

20.1. Apache Qpid JNDI Properties for AMQP Messaging

Apache Qpid supports the following JNDI properties:

connectionfactory.<*jndiname*>
: The Connection URL that the connection factory uses to perform connections.

queue.<*jndiname*>
: A JMS queue. Implemented as an `amq.direct` exchange in Apache Qpid.

topic.<*jndiname*>
: A JMS topic. 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 (or a binding URL, for backward-compatibility with earlier implementations).

Report a bug

20.2. JNDI Properties for Apache Qpid

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

```
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
```

Report a bug

20.3. Connection URLs

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

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

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

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

Apache Qpid supports the following properties in Connection URLs:

Table 20.1. Connection URL Properties

Option	Type	Description
`brokerlist`	Section 20.3, " Broker list URL "	The broker to use for this connection. In the current release, precisely one broker must be specified.
`maxprefetch`	Integer	The maximum number of pre-fetched messages per destination.
`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' | 'singlebroker'}`	• 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. • If singlebroker is selected it connects to the initial broker only and does not support failover.

Broker list URL

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 URL:

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

The following broker list URL options are supported:

Table 20.2. Broker List URL 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	String	Path to Kerberos trust store
trust_store_password	String	Kerberos trust store password
key_store	String	Path to Kerberos key store
key_store_password	String	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`	String	If multiple certificates are present in the keystore, the alias will be used to extract the correct certificate.

Report a bug

20.4. 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 20.3. Java JMS Mapping to AMQP 0-10 Message Properties

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

[a] This is a custom JMS property, set automatically by the Java JMS client implementation.
[b] The reply_to is converted from the protocol representation into an address.
[c] JMSExpiration = dp.ttl + currentTime

Report a bug

20.5. 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 20.1. 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 20.4. 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.lang.String	unicode	std::string
java.util.UUID	uuid	qpid::types::Uuid
java.util.Map [a]	dict	Variant::Map
java.util.List	list	Variant::List

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

Report a bug

20.6. JMS ListMessage

MRG 2.3 introduces a JMS `ListMessage` type.

On the receiver side, List messages are exposed via 3 interfaces:

1. `javax.jms.StreamMessage`
2. `javax.jms.MapMessage`
3. `org.apache.qpid.jms.ListMessage`

On the sender side, List messages can be sent two ways:

1. `org.apache.qpid.jms.ListMessage` - by creating it via `createListMessage()` in `org.apache.qpid.jms.Session`.
   Example:

   ```
   ListMessage msg =  ((org.apache.qpid.jms.Session)ssn).createListMessage();
   ```

2. If you set `-Dqpid.use_legacy_stream_message=false` any stream message you create will be encoded as a list message.
   Example:

   ```
   StreamMessage msg = jmsSession.createStreamMessage();
   ```

For code examples, refer to this sample code.

Report a bug

20.7. JMS Client Logging

The JMS Client logging is handled using the Simple Logging Facade for Java (SLF4J). SLF4J is a facade that delegates to other logging systems like log4j or JDK 1.4 logging.

When using the log4j binding, set the log level for `org.apache.qpid`. 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 20.2. 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
```

Report a bug

20.8. JMS Client Configuration

20.8.1. Configuration Methods and Granularity

The Qpid JMS Client allows several configuration options to customize its behavior at different levels of granularity.

- JVM level using JVM arguments - Affects all connections, sessions, consumers and producers created within the JVM.

  Example: The `-dmax_prefetch=1000` property specifies the message credits to use.
- Connection level using connection or broker properties - Affects the respective connection and sessions, consumers and produces created by that connection.

  Example: The `amqp://guest:guest@test/test?max_prefetch='1000' &brokerlist='tcp://localhost:5672'` property specifies the message credits to use. This overrides any value specified via the JVM argument `max_prefetch`.
- Destination level using addressing options - Affects the producer(s) and consumer(s) created using the respective destination.

  Example: `my-queue; {create: always, link:{capacity: 10}}` where capacity option specifies the message credits to use. This overrides any connection level configuration.

Report a bug

20.8.2. Qpid JVM Arguments

Table 20.5. Configuration Options For Connection Behavior

Property Name	Type	Default Value	Description
`qpid.amqp.version`	string	0-10	Sets the AMQP version to be used - currently supports 0-8, 0-9, 0-91, and 0-10. The client will begin negotiation at the specified version and only negotiate downwards if the broker does not support the specified version.
`qpid.heartbeat`	int	120 (seconds)	The heartbeat interval in seconds. Two consective misssed heartbeats will result in the connection timing out. This can also be set per connection.
`ignore_setclientID`	boolean	false	If a client ID specified in the connection URL it is used, otherwise an ID is generated. If an ID is specified after it has been generated Qpid will throw an exception. Setting this property to 'true' disables that check and allows you to set a client ID at any time.

Table 20.6. Configuration Options For Session Behavior

Property Name	Type	Default Value	Description
`qpid.session.command_limit`	int	65536	Limits the number of unacknowledged commands.
`qpid.session.byte_limit`	int	1048576	Limits the number of unacknowledged commands in bytes.
`qpid.use_legacy_map_message`	boolean	false	Uses the old map message encoding. By default the map messages are encoded using the 0-10 map encoding. This can also be set per connection as well.

Table 20.7. Configuration Options For Consumer Behavior

Property Name	Type	Default Value	Description
`max_prefetch`	int	500	Maximum number of messages to credits. Can also be set per connection or per destination.
`qpid.session.max_ack_delay`	long	1000 (ms)	Timer interval to flush message acks in buffer when using *AUTO_ACK* and *DUPS_OK*.
`sync_ack`	boolean	false	If set, each message will be acknowledged synchronously. When using *AUTO_ACK* mode, set this to "true". Can also be set per connection.

Table 20.8. Configuration Options For Producer Behavior

Property Name	Type	Default Value	Description
`sync_publish`	string	-	Sends messages synchronously. Valid values are *`persistent`* or *`all`*. Can also be set per connection.

Table 20.9. Configuration Options For Threading

Property Name	Type	Default Value	Description
`qpid.thread_factory`	string	`org.apache.qpid.thread.DefaultThreadFactory`	Specifies the thread factory to use. If using a real time JVM, set to `org.apache.qpid.thread.RealtimeThreadFactory`.
`qpid.rt_thread_priority`	int	20	Specifies the priority (1-99) for realtime threads created by the realtime thread factory.

Table 20.10. Configuration Options For I/O

Property Name	Type	Default Value	Description
`qpid.transport`	string	`org.apache.qpid.transport.network.io.IoNetworkTransport`	The transport implementation to be used. You can also specify the `org.apache.qpid.transport.network.NetworkTransport` transport mechanism.
`qpid.sync_op_timeout`	long	60000 (milliseconds)	The length of time to wait for a synchronous operation to complete. For compatibility with older clients, use `amqj.default_syncwrite_timeout`.
`amqj.tcp_nodelay`	boolean	false	Sets the `TCP_NODELAY` property of the underlying socket. Can also be set per connection.

Table 20.11. Configuration Options For Security

Property Name	Type	Default Value	Description
`qpid.sasl_mechs`	string	PLAIN	The SASL mechanism used. More than one can be specified using a comma separated list. Supported values are PLAIN, GSSAPI, and EXTERNAL.
`qpid.sasl_protocol`	string	AMQP	When using GSSAPI as the SASL mechanism, `sasl_protocol` must be set to the principal for the qpidd broker.
`qpid.sasl_server_name`	string	localhost	When using GSSAPI as the SASL mechanism, `sasl_server` must be set to the host for the SASL server.

Table 20.12. JVM properties for GSSAPI as the SASL mechanism

Property Name	Type	Default Value	Description
`javax.security.auth.useSubjectCredsOnly`	boolean	true	If set to 'false', forces the SASL GASSPI client to obtain kerberos credentials explicitly.
`java.security.auth.login.config`	string	-	Specifies the JASS configuration file.

Table 20.13. Configuration options for SSL connections

Property Name	Type	Default Value	Description
`qpid.ssl_timeout`	long	60000	Timeout value used by the Java SSL engine when waiting on operations.
`qpid.ssl.keyStoreCertType`		SunX509	The certificate type.
`qpid.ssl.trustStoreCertType`	string	SunX509	The certificate type.

Table 20.14. JVM Properties for SSL connections

Property Name	Type	Default Value	Description
`javax.net.ssl.keyStore`	string	jvm default	Specifies the key store path.
`javax.net.ssl.keyStorePassword`	string	jvm default	Specifies the key store password.
`javax.net.ssl.trustStore`	string	jvm default	Specifies the trust store path.
`javax.net.ssl.trustStorePassword`	string	jvm default	Specifies the trust store password.

Report a bug

Chapter 21. .NET Binding for Qpid C++ Messaging

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

Table 21.1. 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.

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.2. 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.

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.3. .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();*
	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;*

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

See Also:

- Section 3.3.3.2, "Windows SDK Contents"

Report a bug

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

Table 21.4. .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
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();*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.5. .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	*public static Duration MINUTE;*
.NET	*public static Duration SECOND;*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.6. .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();*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.7. .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	*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	*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	*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);*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.8. .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	*!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
C++	*const std::string& getName() const;*
.NET	*public string Name { get; }*
	Property: Session
C++	*Session getSession() const;*
.NET	*public Session Session { get; }*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.9. .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; }*
	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; }*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

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

Table 21.10. .NET Binding for the C++ Messaging API 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
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
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();*

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

21.11. .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`.

See Also:

- Section 3.3.3.2, “Windows SDK Contents”

Report a bug

Exchange and Queue Declaration Arguments

A.1. Exchange and Queue Argument Reference

Following is a complete list of arguments for declaring queues and exchanges.

Exchange options

`qpid.exclusive-binding` **(bool)**
: Ensures that a given binding key is associated with only one queue.

`qpid.ive` **(bool)**
: If set to “true”, the exchange is an *initial value exchange*, which differs from other exchanges in only one way: the last message sent to the exchange is cached, and if a new queue is bound to the exchange, it attempts to route this message to the queue, if the message matches the binding criteria. This allows a new queue to use the last received message as an initial value.

`qpid.msg_sequence` **(bool)**
: If set to “true”, the exchange inserts a sequence number named “qpid.msg_sequence” into the message headers of each message. The type of this sequence number is int64. The sequence number for the first message routed from the exchange is 1, it is incremented sequentially for each subsequent message. The sequence number is reset to 1 when the qpid broker is restarted.

`qpid.sequence_counter` **(int64)**
: Start `qpid.msg_sequence` counting at the given number.

Queue options

`no-local` **(bool)**
: Specifies that the queue should discard any messages enqueued by sessions on the same connection as that which declares the queue.

`qpid.alert_count` **(uint32_t)**
: If the queue message count goes above this size an alert should be sent.

`qpid.alert_repeat_gap` **(int64_t)**
: Controls the minimum interval between events in seconds. The default value is 60 seconds.

`qpid.alert_size` **(int64_t)**
: If the queue size in bytes goes above this size an alert should be sent.

`qpid.auto_delete_timeout` **(bool)**
: If a queue is configured to be automatically deleted, it will be deleted after the amount of seconds specified here.

`qpid.browse-only` **(bool)**
All users of queue are forced to browse. Limit queue size with ring, LVQ, or TTL. Note that this argument name uses a hyphen rather than an underscore.

`qpid.file_count` **(int)**
Set the number of files in the persistence journal for the queue. Default value is 8.

`qpid.file_size` **(int64)**
Set the number of pages in the file (each page is 64KB). Default value is 24.

`qpid.flow_resume_count` **(uint32_t)**
Flow resume threshold value as a message count.

`qpid.flow_resume_size` **(uint64_t)**
Flow resume threshold value in bytes.

`qpid.flow_stop_count` **(uint32_t)**
Flow stop threshold value as a message count.

`qpid.flow_stop_size` **(uint64_t)**
Flow stop threshold value in bytes.

`qpid.last_value_queue` **(bool)**
Enables last value queue behavior.

`qpid.last_value_queue_key` **(string)**
Defines the key to use for a last value queue.

`qpid.last_value_queue_no_browse` **(bool)**
Enables special mode for last value queue behavior.

`qpid.max_count` **(uint32_t)**
The maximum byte size of message data that a queue can contain before the action dictated by the `policy_type` is taken.

`qpid.max_size` **(uint64_t)**
The maximum number of messages that a queue can contain before the action dictated by the `policy_type` is taken.

`qpid.msg_sequence` **(bool)**
Causes a sequence number to be added to headers of enqueued messages.

`qpid.optimistic_consume` **(bool)**
Allows the consumer to dequeue the message before the broker has acknowledged the producer, in order to reduce latency for durable messaging.

`qpid.persist_last_node` **(bool)**
Allows for a queue to treat all transient messages as persistent when a cluster fails down

to a single node. When additional nodes in the cluster are restored, the transient messages will no longer be persisted. This mode will not be triggered if a cluster is started with only one active node, and the queues in this mode must be configured to be durable.

`qpid.policy_type` **(string)**
Sets default behavior for controlling queue size. Valid values are *`reject`*, *`flow_to_disk`*, *`ring`*, and *`ring_strict`*.

`qpid.priorities` **(size_t)**

The number of distinct priority levels recognized by the queue (up to a maximum of 10). The default value is 1 level.

`qpid.queue_event_generation` **(type: int)**

If the queue is created within a program, sets the queue options to enable queue events. Use the value `1` to replicate only enqueue events, or `2` to replicate both enqueue and dequeue events.

`qpid.trace.exclude` **(string)**
Does not send on messages which include one of the given (comma separated) trace ids.

`qpid.trace.id` **(string)**
Adds the given trace id as to the application header "`x-qpid.trace`" in messages sent from the queue.

`x-qpid-maximum-message-count`
This is an alias for `qpid.alert_count`.

`x-qpid-maximum-message-size`
This is an alias for `qpid.alert_size`.

`x-qpid-minimum-alert-repeat-gap`
This is an alias for `qpid.alert_repeat_gap`.

`x-qpid-priorities`
This is an alias for `qpid.priorities`.

Report a bug

Changes

B.1. New for 2.3

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

See Also:

- Section 1.5, “Differences between AMQP 0-10 and AMQP 1.0”
- Section 9.13, “ACL Lookup Query Methods”
- Section 18.3, “SASL Support in Windows Clients”
- Section 9.11, “QMF Events”
- Section 9.12, “QMF Client Connection Events”
- Section 10.1.4, “Session Exceptions”
- Section 20.6, “JMS ListMessage”

Report a bug

Revision History

Revision 2.0.0-31	**Fri Feb 22 2013**	**Joshua Wulf**
Built from Content Specification: 8025, Revision: 373300 by jwulf		