Programming in Apache Qpid

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

Programming in Apache Qpid: Cross-Platform AMQP Messaging in Java JMS, .NET, C++, and Python

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

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

- On the Java platform, Qpid uses the established Java JMS API [http://java.sun.com/products/jms/].
- On the .NET platform, Qpid defines a WCF binding [http://qpid.apache.org/wcf.html].
- For Python, C++, and .NET, Qpid defines its own messaging API, the *Qpid Messaging API*, which is conceptually similar in each supported language.
- Support for this API in Ruby will be added soon (Ruby currently uses an API that is closely tied to the AMQP version).

Chapter 2. Using the Qpid Messaging API

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

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

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

1. A Simple Messaging Program in C++

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

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

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

#include <iostream>

using namespace qpid::messaging;

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

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

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

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

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

1 Establishes the connection with the messaging broker.

2 Creates a session object, which maintains the state of all interactions with the messaging broker, and manages senders and receivers.

3 Creates a receiver that reads from the given address.

4 Creates a sender that sends to the given address.

5 Reads the next message. The duration is optional, if omitted, will wait indefinitely for the next message.

6 Acknowledges messages that have been read. To guarantee delivery, a message remains on the messaging broker until it is acknowledged by a client. session.acknowledge() acknowledges all unacknowledged messages for the given session—this allows acknowledgements to be batched, which is more efficient than acknowledging messages individually.

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

2. A Simple Messaging Program in Python

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

Example 2.2. "Hello world!" in Python

```
import sys
from qpid.messaging import *

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

connection = Connection(broker)

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

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

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

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

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

1. Establishes the connection with the messaging broker.
2. Creates a session object, which maintains the state of all interactions with the messaging broker, and manages senders and receivers.
4. Creates a receiver that reads from the given address.
3. Creates a sender that sends to the given address.
5. Reads the next message. The duration is optional, if omitted, will wait indefinitely for the next message.
6. Acknowledges messages that have been read. To guarantee delivery, a message remains on the messaging broker until it is acknowledged by a client. session.acknowledge() acknowledges all unacknowledged messages for the given session—this allows acknowledgements to be batched, which is more efficient than acknowledging messages individually.
7. Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

3. A Simple Messaging Program in .NET C#

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

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

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

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

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

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

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

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

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

1. Selects the Qpid Messaging namespace. A project reference to the Org.Apache.Qpid.Messaging dll defines the Qpid Messaging namespace objects and methods.
2. Establishes the connection with the messaging broker.
3. Creates a session object, which maintains the state of all interactions with the messaging broker, and manages senders and receivers.
4. Creates a receiver that reads from the given address.
5. Creates a sender that sends to the given address.
6. Reads the next message. The duration is optional, if omitted, will wait indefinitely for the next message.
7. Acknowledges messages that have been read. To guarantee delivery, a message remains on the messaging broker until it is acknowledged by a client. session.acknowledge() acknowledges all unacknowledged messages for the given session—this allows acknowledgements to be batched, which is more efficient than acknowledging messages individually.
8. Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.

4. Addresses

An *address* is the name of a message target or message source. In the programs we have just seen, we used the address `amq.topic` (which is the name of an exchange on an AMQP 0-10 messaging broker). The methods that create senders and receivers require an address. The details of sending to a particular target or receiving from a particular source are then handled by the sender or receiver. A different target or source can be used simply by using a different address.

An address resolves to a *node*. The Qpid Messaging API recognises two kinds of nodes, *queues* and *topics* [1]. A queue stores each message until it has been received and acknowledged, and only one receiver can receive a given message [2]. A topic immediately delivers a message to all eligible receivers; if there are no eligible receivers, it discards the message. In the AMQP 0-10 implementation of the API, [3] queues map to AMQP queues, and topics map to AMQP exchanges. [4]

In the rest of this tutorial, we present many examples using two programs that take an address as a command line parameter. **spout** sends messages to the target address, **drain** receives messages from the source address. The source code is available in C++, Python, and .NET C# and can be found in the examples directory for each language. These programs can use any address string as a source or a destination, and have many command line options to configure behavior—use the **-h** option for documentation on these options. [5] The examples in this tutorial also use the **qpid-config** utility to configure AMQP 0-10 queues and exchanges on a Qpid broker.

Example 2.4. Queues

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

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

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

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

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

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

[1] The terms *queue* and *topic* here were chosen to align with their meaning in JMS. These two addressing 'patterns', queue and topic, are sometimes refered as point-to-point and publish-subscribe. AMQP 0-10 has an exchange type called a *topic exchange*. When the term *topic* occurs alone, it refers to a Messaging API topic, not the topic exchange.

[2] There are exceptions to this rule; for instance, a receiver can use `browse` mode, which leaves messages on the queue for other receivers to read.

[3] The AMQP 0-10 implementation is the only one that currently exists.

[4] In AMQP 0-10, messages are sent to exchanges, and read from queues. The Messaging API also allows a sender to send messages to a queue; internally, Qpid implements this by sending the message to the default exchange, with the name of the queue as the routing key. The Messaging API also allows a receiver to receive messages from a topic; internally, Qpid implements this by setting up a private subscription queue for the receiver and binding the subscription queue to the exchange that corresponds to the topic.

[5] Currently, the C++, Python, and .NET C# implementations of **drain** and **spout** have slightly different options. This tutorial uses the C++ implementation. The options will be reconciled in the near future.

Example 2.5. Topics

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

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

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

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

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

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

Now let's run **drain** first, using the `-t` option to specify a timeout in seconds. While **drain** is waiting for messages, run **spout** in another window.

First Window:

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

Second Window:

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

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

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

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

4.1. Address Strings

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

The syntax for an address string is:

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

Addresses, subjects, and keys are strings. Values can be numbers, strings (with optional single or double quotes), maps, or lists. A complete BNF for address strings appears in Section 4.4, “Address String Grammar”.

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

4.2. Subjects

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

In AMQP 0-10, each exchange type has its own matching algorithm, and queues do not provide filtering. This is discussed in Section 13, “The AMQP 0-10 mapping”.

Note

Currently, a receiver bound to a queue ignores subjects, receiving messages from the queue without filtering. In the future, if a receiver is bound to a queue, and its address contains a subject, the subject will be used as a selector to filter messages.

Example 2.6. Using subjects

In this example we show how subjects affect message flow.

First, let's use **qpid-config** to create a topic exchange.

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

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

First Window:

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

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

Second Window:

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

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

```
Message(properties={qpid.subject:sports, spout-id:9441674e-a157-4780-a78e-f7ccea99
```

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

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

Example 2.7. Subjects with multi-word keys

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

Let's use **drain** with the subject `*.news` to listen for messages in which the second word of the key is `news`.

First Window:

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

Now let's send messages using several different two-word keys:

Second Window:

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

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

```
Message(properties={qpid.subject:usa.news, spout-id:73fc8058-5af6-407c-9166-b49a90
Message(properties={qpid.subject:europe.news, spout-id:f72815aa-7be4-4944-99fd-c64
```

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

First Window:

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

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

Second Window:

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

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

```
Message(properties={qpid.subject:news, spout-id:cbd42b0f-c87b-4088-8206-26d7627c96
Message(properties={qpid.subject:usa.news, spout-id:234a78d7-daeb-4826-90e1-1c6540
Message(properties={qpid.subject:usa.faux.news, spout-id:6029430a-cfcb-4700-8e9b-c
```

4.3. Address String Options

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

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

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

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

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

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

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

- Extension points that rely on the functionality of specific node types.

  For instance, the Qpid XML exchange can use XQuery to do content-based routing for XML messages, or to query message data using XQuery. Queries can be specified using options.

Let's use some examples to show how these different kinds of address string options affect the behavior of senders and receives.

4.3.1. assert

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

Example 2.8. Assertions on Nodes

Let's use **qpid-config** to create a queue and a topic.

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

We can now use the address specified to drain to assert that it is of a particular type:

```
$ ./drain 'my-queue; {assert: always, node:{ type: queue }}'
$ ./drain 'my-queue; {assert: always, node:{ type: topic }}'
2010-04-20 17:30:46 warning Exception received from broker: not-found: not-found:
Exchange my-queue does not exist
```

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

We can do the same thing for my-topic:

```
$ ./drain 'my-topic; {assert: always, node:{ type: topic }}'
$ ./drain 'my-topic; {assert: always, node:{ type: queue }}'
2010-04-20 17:31:01 warning Exception received from broker: not-found: not-found:
Queue my-topic does not exist
```

Now let's use the `create` option to create the queue `xoxox` if it does not already exist:

4.3.2. create

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

Example 2.9. Creating a Queue Automatically

First Window:

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

Now we can send messages to this queue:

Second Window:

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

Returning to the first window, we see that **drain** has received this message:

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

4.3.3. browse

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

Example 2.10. Browsing a Queue

Let's use the browse mode to receive messages without removing them from the queue. First we send three messages to the queue:

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

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

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

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

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

4.3.4. x-bindings

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

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

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

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

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

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

Example 2.11. Using the XML Exchange

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

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

We can specify this query in an x-binding to listen to messages that meet the criteria specified by the query:

First Window:

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

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

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

Now let's use **spout** to send this message to the XML exchange:

Second Window:

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

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

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

4.3.5. Address String Options - Reference

Table 2.1. Address String Options

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

Table 2.2. Node Properties

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

property	value	semantics

	fields that describe an AMQP 0-10 binding. Here is the format for x-bindings:
	
	`[`
	`{`
	`exchange: <exchange>,`
	`queue: <queue>,`
	`key: <key>,`
	`arguments: {`
	`<key_1>: <value_1>,`
	`...,`
	`<key_n>: <value_n> }`
	`},`
	`...`
	`]`
		with the assert option, the existence of each of these bindings is verified during resolution. Again, these are protocol specific.

Table 2.3. Link Properties

option	value	semantics
reliability	one of: unreliable, at-least-once, at-most-once, exactly-once	Reliability indicates the level of reliability that the sender or receiver. `unreliable` and `at-most-once` are currently treated as synonyms, and allow messages to be lost if a broker crashes or the connection to a broker is lost. `at-least-once` guarantees that a message is not lost, but duplicates may be received. `exactly-once` guarantees that a message is not lost, and is delivered precisely once.
durable	True, False	Indicates whether the link survives a loss of volatile storage e.g. if the broker is restarted.
x-declare	A nested map whose values correspond to the valid fields of an AMQP 0-10 queue-declare command.	These values can be used to customise the subscription queue in the case of receiving from an exchange. Note however that they are protocol specific.
x-subscribe	A nested map whose values correspond to the valid fields of an AMQP 0-10 message-subscribe command.	These values can be used to customise the subscription.
x-bindings	A nested list each of whose entries is a map that may contain fields (queue, exchange, key and arguments) describing an AMQP 0-10 binding.	These bindings are established during resolution independent of the create option. They are considered logically part of the linking process rather than of node creation.

4.4. Address String Grammar

This section provides a formal grammar for address strings.

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

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

Grammar. The formal grammar for addresses is given below:

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

Address String Options. The address string options map supports the following parameters:

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

```
    reliability: unreliable | at-most-once | at-least-once | exactly-once,
    x-declare: { ... <declare-overrides> ... },
    x-bindings: [<binding_1>, ... <binding_n>],
    x-subscribe: { ... <subscribe-overrides> ... }
  }
}
```

Create, Delete, and Assert Policies

The create, delete, and assert policies specify who should perfom the associated action:

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

Node-Type

The node-type is one of:

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

5. Logging

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

5.1. Logging in C++

The Qpidd broker and C++ clients can both use environment variables to enable logging. Use QPID_LOG_ENABLE to set the level of logging you are interested in (trace, debug, info, notice, warning, error, or critical):

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

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

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

5.2. Logging in Python

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

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

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

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

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

6. Receiving Messages from Multiple Sources

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

Example 2.12. Receiving Messages from Multiple Sources

C++:

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

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

Python:

```
receiver1 = session.receiver(address1)
receiver2 = session.receiver(address)
message = session.next_receiver().fetch()
print message.content
```

.NET C#:

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

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

7. Request / Response

Request / Response applications use the reply-to property, described in Table 2.8, "Mapping to AMQP 0-10 Message Properties", to allow a server to respond to the client that sent a message. A server sets up

a service queue, with a name known to clients. A client creates a private queue for the server's response, creates a message for a request, sets the request's reply-to property to the address of the client's response queue, and sends the request to the service queue. The server sends the response to the address specified in the request's reply-to property.

Example 2.13. Request / Response Applications in C++

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

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

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

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

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

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

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

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

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

8. Maps in Message Content

Many messaging applications need to exchange data across languages and platforms, using the native datatypes of each programming language. AMQP provides a set of portable datatypes, but does not directly support a set of named type/value pairs. Java JMS provides the `MapMessage` interface, which allows sets of named type/value pairs, but does not provide a set of portable datatypes.

The Qpid Messaging API supports maps in message content. Unlike JMS, any message can contain maps. These maps are supported in each language using the conventions of the language. In Java, we implement the `MapMessage` interface; in Python, we support `dict` and `list` in message content; in C++, we

provide the `Variant::Map` and `Variant::List` classes to represent maps and lists. In all languages, messages are encoded using AMQP's portable datatypes.

Tip

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

8.1. Qpid Maps in Python

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

Example 2.14. Sending Qpid Maps in Python

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

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

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

Table 2.4. Python Datatypes in Maps

Python Datatype	**# 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

8.2. Qpid Maps in C++

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

Example 2.15. Sending Qpid Maps in C++

```
using namespace qpid::types;

// !!! SNIP !!!

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

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

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

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

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

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

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

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

Table 2.5. C++ Datatypes in Maps

C++ Datatype	# 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

9. Performance

Clients can often be made significantly faster by batching acknowledgements and setting the capacity of receivers to allow prefetch. The size of a sender's replay buffer can also affect performance.

9.1. Batching Acknowledgements

Many of the simple examples we have shown retrieve a message and immediately acknowledge it. Because each acknowledgement results in network traffic, you can dramatically increase performance by acknowledging messages in batches. For instance, an application can read a number of related messages, then acknowledge the entire batch, or an application can acknowledge after a certain number of messages have been received or a certain time period has elapsed. Messages are not removed from the broker until they are acknowledged, so guaranteed delivery is still available when batching acknowledgements.

9.2. Prefetch

By default, a receiver retrieves the next message from the server, one message at a time, which provides intuitive results when writing and debugging programs, but does not provide optimum performance. To create an input buffer, set the capacity of the receiver to the size of the desired input buffer; for many applications, a capacity of 100 performs well.

Example 2.16. Prefetch

C++

```
Receiver receiver = session.createReceiver(address);
receiver.setCapacity(100);
Message message = receiver.fetch();
```

9.3. Sizing the Replay Buffer

In order to guarantee delivery, a sender automatically keeps messages in a replay buffer until the messaging broker acknowledges that they have been received. The replay buffer is held in memory, and is never paged to disk. For most applications, the default size of the replay buffer works well. A large replay buffer requires more memory, a small buffer can slow down the client because it can not send new messages if the replay buffer is full, and must wait for existing sends to be acknowledged.

Example 2.17. Sizing the Replay Buffer

C++

```
Sender sender = session.createSender(address);
sender.setCapacity(100);
```

10. Reliability

The Qpid Messaging API supports automatic reconnect, guaranteed delivery via persistent messages, reliability options in senders and receivers, and cluster failover. This section shows how programs can take advantage of these features.

10.1. Reconnect

Connections in the Qpid Messaging API support automatic reconnect if a connection is lost. This is done using connection options. The following example shows how to use connection options in C++ and Python.

Example 2.18. Specifying Connection Options in C++ and Python

In C++, these options are set using `Connection::setOption()`:

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

In Python, these options are set using named arguments in the `Connection` constructor:

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

See the reference documentation for details on how to set these on connections for each language.

The following table lists the connection options that can be used.

Table 2.6. Connection Options

option	value	semantics
reconnect	True, False	Transparently reconnect if the connection is lost.
reconnect_timeout	N	Total number of seconds to continue reconnection attempts before giving up and raising an exception.
reconnect_limit	N	Maximum number of reconnection attempts before giving up and raising an exception.
reconnect_interval_min	N	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	N	Maximum reconnect interval.
reconnect_interval	N	Sets both `reconnection_interval_min` and `reconnection_interval_max` to the same value.

10.2. Guaranteed Delivery

If a queue is durable, the queue survives a messaging broker crash, as well as any durable messages that have been placed on the queue. These messages will be delivered when the messaging broker is restarted. Delivery is guaranteed if and only if both the message and the queue are durable. Guaranteed delivery requires a persistence module, such as the one available from QpidComponents.org [http://QpidComponents.org].

Example 2.19. Guaranteed Delivery

C++:

```
Sender sender = session.createSender("durable-queue");

Message message("Hello world!");
message.setDurable(1);

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

10.3. Reliability Options in Senders and Receivers

When creating a sender or a receiver, you can specify a reliability option in the address string. For instance, the following specifies `at-least-once` as the reliability mode for a sender:

```
Sender = session.createSender("topic;{create:always,link:{reliability:at-least-onc
```

The modes `unreliable`, `at-most-once`, `at-least-once`, and `exactly-once` are supported. These modes govern the reliability of the connection between the client and the messaging broker.

The modes `unreliable` and `at-most-once` are currently synonyms. In a receiver, this mode means that messages received on an auto-delete subscription queue may be lost in the event of a broker failure. In a sender, this mode means that the sender can consider a message sent as soon as it is written to the wire, and need not wait for broker acknowledgement before considering the message sent.

The mode `at-most-once` ensures that messages are not lost, but duplicates of a message may occur. In a receiver, this mode ensures that messages are not lost in event of a broker failure. In a sender, this means that messages are kept in a replay buffer after they have been sent, and removed from this buffer only after the broker acknowledges receipt; if a broker failure occurs, messages in the replay buffer are resent upon reconnection. The mode `exactly-once` is similar to `at-most-once`, but eliminates duplicate messages.

10.4. Cluster Failover

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

In C++, the `FailoverUpdates` class keeps track of the brokers in a cluster, so a reconnect can select another broker in the cluster to connect to:

Example 2.20. Cluster Failover in C++

```
#include <qpid/messaging/FailoverUpdates.h>
...
Connection connection(broker);
connection.setOption("reconnect", true);
try {
    connection.open();
    std::auto_ptr<FailoverUpdates> updates(new FailoverUpdates(connection));
```

11. Security

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

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

To enable Kerberos in a client, set the `sals-mechanism` connection option to `GSSAPI`:

```
Connection connection(broker);
connection.setOption("sasl-mechanism", "GSSAPI");
try {
    connection.open();
    ...
```

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

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

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

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

Use the following environment variables to configure the SSL client:

Table 2.7. 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.

12. Transactions

In AMQP, transactions cover the semantics of enqueues and dequeues.

When sending messages, a transaction tracks enqueues without actually delivering the messages, a commit places messages on their queues, and a rollback discards the enqueues.

When receiving messages, a transaction tracks dequeues without actually removing acknowledged messages, a commit removes all acknowledged messages, and a rollback discards acknowledgements. A rollback does not release the message, it must be explicitly released to return it to the queue.

Example 2.21. Transactions

C++:

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

13. The AMQP 0-10 mapping

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

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

When sending to a queue, the queue's name is set as the routing key and the message is transfered to the default (or nameless) exchange. When sending to an exchange, the message is transfered 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,[6] if a subject is specified it is used as the binding key and an XQuery is defined that matches any message with that value for qpid.subject. Again this means that only messages whose subject exactly match that specified in the source address are received. If no subject is specified then the empty string is used as the binding key with an xquery that will match any message (this means that only messages with an empty string as the routing key will be received). For more control the x-bindings element in the link properties must be used. A source address that resolves to the XML exchange must contain either a subject or an x-bindings element in the link properties as there is no way at present to receive any message regardless of routing key.

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

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

Table 2.8. Mapping to AMQP 0-10 Message Properties

Python API	**C++ API**	**AMQP 0-10 Property**[a]
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[b]
msg.correlation_id	msg.{get,set}CorrelationId()	mp.correlation_id
msg.durable	msg.{get,set}Durable()	dp.delivery_mode == delivery_mode.persistent[c]
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] In these entries, `mp` refers to an AMQP message property, and `dp` refers to an AMQP delivery property.

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

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

Chapter 3. Using the Qpid JMS client

1. A Simple Messaging Program in Java JMS

The following program shows how to use address strings and JNDI for Qpid programs that use Java JMS.

The Qpid JMS client uses Qpid Messaging API Section 4, “Addresses” to identify sources and targets. This program uses a JNDI properties file that defines a connection factory for the broker we are using, and the address of the topic exchange node that we bind the sender and receiver to. (The syntax of a ConnectionURL is given in Section 2, “Apache Qpid JNDI Properties for AMQP Messaging”.)

Example 3.1. JNDI Properties File for "Hello world!" 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
```

In the Java JMS code, we use create a JNDI context, use the context to find a connection factory and create and start a connection, create a session, and create a destination that corresponds to the topic exchange. Then we create a sender and a receiver, send a message with the sender, and receive it with the receiver. This code should be straightforward for anyone familiar with Java JMS.

Example 3.2. "Hello world!" in Java

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

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

public class Hello {

  public Hello() {
  }

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

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

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

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

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

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

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

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

1. Loads the JNDI properties file, which specifies connection properties, queues, topics, and addressing options. See Section 2, “Apache Qpid JNDI Properties for AMQP Messaging” for details.
2. Creates the JNDI initial context.
3. Creates a JMS connection factory for Qpid.
4. Creates a JMS connection.
5. Activates the connection.
6. Creates a session. This session is not transactional (transactions='false'), and messages are automatically acknowledged.
7. Creates a destination for the topic exchange, so senders and receivers can use it.
8. Creates a producer that sends messages to the topic exchange.
9. Creates a consumer that reads messages from the topic exchange.
10. Reads the next available message.
11. Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.
12. Closes the JNDI context.

2. Apache Qpid JNDI Properties for AMQP Messaging

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

Example 3.3. JNDI Properties File

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

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

The following sections describe the JNDI properties that Qpid uses.

2.1. JNDI Properties for Apache Qpid

Apache Qpid supports the properties shown in the following table:

Table 3.1. JNDI Properties supported by Apache Qpid

Property	Purpose
connectionfactory.<jndiname>	The Connection URL that the connection factory uses to perform connections.
queue.<jndiname>	A JMS queue, which is implemented as an amq.direct exchange in Apache Qpid.
topic.<jndiname>	A JMS topic, which is implemented as an amq.topic exchange in Apache Qpid.

Property	Purpose
destination.<jndiname>	Can be used for defining all amq destinations, queues, topics and header matching, using an address string. [a]

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

2.2. Connection URLs

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

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

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

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

Apache Qpid supports the following properties in Connection URLs:

Table 3.2. Connection URL Properties

Option	Type	Description
brokerlist	see below	The broker to use for this connection. In the current release, precisely one broker must be specified.
maxprefetch	--	The maximum number of pre-fetched messages per destination.
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 acknowledgement to guarantee that it has been received.
use_legacy_map_msg_format	Boolean	If you are using JMS Map messages and deploying a new client with any JMS client older than 0.7 release, you must set this to true to ensure the older clients can understand the map message encoding.
failover	{'roundrobin' | 'failover_exchange'}	If roundrobin is selected it will try each broker given in the broker list. If failover_exchange is selected it connects to the initial broker given in the broker URL and will receive membership updates via the failover exchange.

Broker lists are specified using a URL in this format:

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

For instance, this is a typical broker list:

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

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

Example 3.4. Broker Lists

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

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

This broker list specifies SSL options:

```
amqp://guest:guest@test/test?sync_ack='true'
   &brokerlist='tcp://ip1:5672?ssl='true'&ssl_cert_alias='cert1'
```

The following broker list options are supported.

Table 3.3. Broker List Options

Option	Type	Description
heartbeat	integer	frequency of heartbeat messages (in seconds)
sasl_mechs	--	For secure applications, we suggest CRAM-MD5, DIGEST-MD5, or GSSAPI. The ANONYMOUS method is not secure. The PLAIN method is secure only when used together with SSL. For Kerberos, sasl_mechs must be set to GSSAPI, sasl_protocol must be set to the principal for the qpidd broker, e.g. qpidd/, and sasl_server must be set to the host for the SASL server, e.g. sasl.com. SASL External is supported using

Option	Type	Description
		SSL certification, e.g. `ssl='true'&sasl_mechs='EXTERNAL'`
sasl_encryption	Boolean	If `sasl_encryption='true'`, the JMS client attempts to negotiate a security layer with the broker using GSSAPI to encrypt the connection. Note that for this to happen, GSSAPI must be selected as the sasl_mech.
ssl	Boolean	If `ssl='true'`, the JMS client will encrypt the connection using SSL.
tcp_nodelay	Boolean	If `tcp_nodelay='true'`, TCP packet batching is disabled.
sasl_protocol	--	Used only for Kerberos. `sasl_protocol` must be set to the principal for the qpidd broker, e.g. `qpidd/`
sasl_server	--	For Kerberos, sasl_mechs must be set to GSSAPI, sasl_server must be set to the host for the SASL server, e.g. `sasl.com`.
trust_store	--	path to Keberos trust store
trust_store_password		Kerberos trust store password
key_store		path to Kerberos key store
key_store_password	--	Kerberos key store password
ssl_verify_hostname	Boolean	When using SSL you can enable hostname verification by using "ssl_verify_hostname=true" in the broker URL.
ssl_cert_alias		If multiple certificates are present in the keystore, the alias will be used to extract the correct certificate.

3. Java JMS Message Properties

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

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

Java JMS Message Property	AMQP 0-10 Property[a]
JMSMessageID	mp.message_id
qpid.subject[b]	mp.application_headers["qpid.subject"]
JMSXUserID	mp.user_id
JMSReplyTo	mp.reply_to[c]

Java JMS Message Property	AMQP 0-10 Property[a]
JMSCorrelationID	mp.correlation_id
JMSDeliveryMode	dp.delivery_mode
JMSPriority	dp.priority
JMSExpiration	dp.ttl[d]
JMSRedelivered	dp.redelivered
JMS Properties	mp.application_headers
JMSType	mp.content_type

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

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

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

[d]JMSExpiration = dp.ttl + currentTime

4. JMS MapMessage Types

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

Example 3.5. Sending a Java JMS MapMessage

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

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

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

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

// !!! SNIP !!!

MessageProducer producer = session.createProducer(queue);

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

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

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

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

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

producer.send(m);
```

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

Table 3.5. Java Datatypes in Maps

Java Datatype	# 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.

5. JMS Client Logging

The JMS Client logging is handled using SLF4J [http://www.slf4j.org/]. A user can place a slf4j binding of their choice in the classpath and configure the respective logging mechanism to suit their needs. Ex bindings include log4j, jdk1.4 logging ..etc

Following is an example on how to configure the client logging with the log4j binding.

For more information on how to configure log4j, please consult the log4j documentation.

If using the log4j binding please ensure to set the log level for org.apache.qpid explicitly. Otherwise log4j will default to DEBUG which will degrade performance considerably due to excessive logging. Recommended logging level for production is WARN.

Example 3.6. log4j Logging Properties

You could place the snippet below in a log4j.properties file and place it in the classpath or explicitly specify it using the -Dlog4j.configuration property. The following configures the qpid client to log at the WARN level

```
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
```

Chapter 4. Using the Qpid WCF client

1. XML and Binary Bindings

The Qpid WCF client provides two bindings, each with support for Windows .NET transactions.

The AmqpBinding is suitable for communication between two WCF applications. By default it uses the WCF binary .NET XML encoder (BinaryMessageEncodingBindingElement) for efficient message transmission, but it can also use the text and Message Transmission Optimization Mechanism (MTOM) encoders. Here is a traditional service model sample program using the AmqpBinding. It assumes that the queue "hello_service_node" has been created and configured on the AMQP broker.

Example 4.1. Traditional service model "Hello world!" example

```
namespace Apache.Qpid.Documentation.HelloService
{
  using System;
  using System.ServiceModel;
  using System.ServiceModel.Channels;
  using System.Threading;
  using Apache.Qpid.Channel;

  [ServiceContract]
  public interface IHelloService
  {
    [OperationContract(IsOneWay = true, Action = "*")]
    void SayHello(string greeting);
  }

  public class HelloService : IHelloService
  {
    private static int greetingCount;

    public static int GreetingCount
    {
      get { return greetingCount; }
    }
    static void Main(string[] args)
    {
      try
    public void SayHello(string greeting)
    { {
      Console.WriteLine("service received: {0}", greeting);
      greetingCount++;
    }   AmqpBinding amqpBinding = new AmqpBinding();
        amqpBinding.BrokerHost = "localhost";
        amqpBinding.BrokerPort = 5672;

        ServiceHost serviceHost = new ServiceHost(typeof(HelloService));
        serviceHost.AddServiceEndpoint(typeof(IHelloService),
          amqpBinding, "amqp:hello_service_node");
        serviceHost.Open();

        // Send the service a test greeting
        Uri amqpClientUri=new Uri("amqp:amq.direct?routingkey=hello_service_node")
        EndpointAddress clientEndpoint = new EndpointAddress(amqpClientUri);
        ChannelFactory<IHelloService> channelFactory =
          new ChannelFactory<IHelloService>(amqpBinding, clientEndpoint);
        IHelloService clientProxy = channelFactory.CreateChannel();

        clientProxy.SayHello("Greetings from WCF client");

        // wait for service to process the greeting
        while (HelloService.GreetingCount == 0)
        {
          Thread.Sleep(100);
        }
        channelFactory.Close();
        serviceHost.Close();
      }
      catch (Exception e)
      {
        Console.WriteLine("Exception: {0}", e);
      }
    }
  }
}
```

The second binding, AmqpBinaryBinding, is suitable for WCF applications that need to inter-operate with non-WCF clients or that wish to have direct access to the raw wire representation of the message body. It relies on a custom encoder to read and write raw (binary) content which operates similarly to the ByteStream encoder (introduced in .NET 4.0). The encoder presents an abstract XML infoset view of the raw message content on input. On output, the encoder does the reverse and peels away the XML infoset layer exposing the raw content to the wire representation of the message body. The application must do the inverse of what the encoder does to allow the XML infoset wrapper to cancel properly. This is demonstrated in the following sample code (using the channel programming model) which directly manipulates or provides callbacks to the WCF message readers and writers when the content is consumed. In contrast to the AmqpBinding sample where the simple greeting is encapsulated in a compressed SOAP envelope, the wire representation of the message contains the raw content and is identical and fully interoperable with the Qpid C++ "Hello world!" example.

```
      port = int.Parse(args[1]);
    }

    if (args.Length > 2)
    {
      target = args[2];
    }
```

Example 4.2. Binary "Hello world!" example using the channel model

```
    if (args.Length > 3)
    {
      source = args[3];
    }

    AmqpBinaryBinding binding = new AmqpBinaryBinding();
    binding.BrokerHost = broker;
    binding.BrokerPort = port;

    IChannelFactory<IInputChannel> receiverFactory = binding.BuildChannelFactory
    receiverFactory.Open();
    IInputChannel receiver = receiverFactory.CreateChannel(new EndpointAddress("
    receiver.Open();

    IChannelFactory<IOutputChannel> senderFactory = binding.BuildChannelFactory<
    senderFactory.Open();
    IOutputChannel sender = senderFactory.CreateChannel(new EndpointAddress("amq
    sender.Open();

    sender.Send(Message.CreateMessage(MessageVersion.None, "", new HelloWorldBin

    Message message = receiver.Receive();
    XmlDictionaryReader reader = message.GetReaderAtBodyContents();
    while (!reader.HasValue)
    {
      reader.Read();
    }

    byte[] binaryContent = reader.ReadContentAsBase64();
    string text = Encoding.UTF8.GetString(binaryContent);

    Console.WriteLine(text);

    senderFactory.Close();
    receiverFactory.Close();
  }
}

public class HelloWorldBinaryBodyWriter : BodyWriter
{
  public HelloWorldBinaryBodyWriter() : base (true) {}

  protected override void OnWriteBodyContents(XmlDictionaryWriter writer)
  {
    byte[] binaryContent = Encoding.UTF8.GetBytes("Hello world!");

    // wrap the content:
    writer.WriteStartElement("Binary");
    writer.WriteBase64(binaryContent, 0, binaryContent.Length);
  }
}
}
```

Bindings define ChannelFactories and ChannelListeners associated with an AMQP Broker. WCF will frequently automatically create and manage the life cycle of a these and the resulting IChannel objects used in message transfer. The binding parameters that can be set are:

Table 4.1. WCF Binding Parameters

Parameter	Default	Description
BrokerHost	localhost	The broker's server name. Currently the WCF channel only supports connections with a single broker. Failover to multiple brokers will be provided in the future.
BrokerPort	5672	The port the broker is listening on.
PrefetchLimit	0	The number of messages to prefetch from the amqp broker before the application actually consumes them. Increasing this number can dramatically increase the read performance in some circumstances.
Shared	false	Indicates if separate channels to the same broker can share an underlying AMQP tcp connection (provided they also share the same authentication credentials).
TransferMode	buffered	Indicates whether the channel's encoder uses the WCF BufferManager cache to temporarily store message content during the encoding/decoding phase. For small to medium sized SOAP based messages, buffered is usually the preferred choice. For binary messages, streamed TransferMode is the more efficient mode.

2. Endpoints

In Qpid 0.6 the WCF Endpoints map to simple AMQP 0-10 exchanges (IOutputChannel) or AMQP 0-10 queues (IInputChannel). The format for an IOutputChannel is

```
"amqp:amq.direct" or "amqp:my_exchange?routingkey=my_routing_key"
```

and for an IInputChannel is

```
"amqp:my_queue"
```

The routing key is in fact a default value associated with the particular channel. Outgoing messages can always have their routing key uniquely set.

If the respective queue or exchange doesn't exist, an exception is thrown when opening the channel. Queues and exchanges can be created and configured using qpid-config.

3. Message Headers

AMQP specific message headers can be set on or retrieved from the ServiceModel.Channels.Message using the AmqpProperties type.

For example, on output:

```
AmqpProperties props = new AmqpProperties();
props.Durable = true;
props.PropertyMap.Add("my_custom_header", new AmqpString("a custom value"));
Message msg = Message.CreateMessage(args);
msg.Properties.Add("AmqpProperties", amqpProperties);
outputChannel.Send(msg);
```

On input the headers can be accessed from the Message or extracted from the operation context

```
public void SayHello(string greeting)
{
  AmqpProperties props = (AmqpProperties) OperationContext.
  Current.IncomingMessageProperties["AmqpProperties"];
  AmqpString extra = (AmqpString) props.PropertyMap["my_custom_header"];
  Console.WriteLine("Service received: {0} and {1}", greeting, extra);
}
```

4. Security

To engage TLS/SSL:

```
binding.Security.Mode = AmqpSecurityMode.Transport;
binding.Security.Transport.UseSSL = true;
binding.BrokerPort = 5671;
```

Currently the WCF client only provides SASL PLAIN (i.e. username and password) authentication. To provide a username and password, you can set the DefaultAmqpCredential value in the binding. This value can be overridden or set for a binding's channel factories and listeners, either by setting the ClientCredentials as a binding parameter, or by using an AmqpCredential as a binding parameter. The search order for credentials is the AmqpCredential binding parameter, followed by the ClientCredentials (unless IgnoreEndpointClientCredentials has been set), and finally defaulting to the DefaultAmqpCredential of the binding itself. Here is a sample using ClientCredentials:

```
ClientCredentials credentials = new ClientCredentials();
credentials.UserName.UserName = "guest";
credentials.UserName.Password = "guest";
bindingParameters = new BindingParameterCollection();
bindingParameters.Add(credentials);
readerFactory = binding.BuildChannelFactory<IInputChannel>(bindingParameters);
```

5. Transactions

The WCF channel provides a transaction resource manager module and a recovery module that together provide distributed transaction support with one-phase optimization. Some configuration is required on Windows machines to enable transaction support (see your installation notes or top level ReadMe.txt file for instructions). Once properly configured, the Qpid WCF channel acts as any other System.Transactions aware resource, capable of participating in explicit or implicit transactions.

Server code:

```
[OperationBehavior(TransactionScopeRequired = true,
                   TransactionAutoComplete = true)]

public void SayHello(string greeting)
{
  // increment ExactlyOnceReceived counter on DB

  // Success: transaction auto completes:
}
```

Because this operation involves two transaction resources, the database and the AMQP message broker, this operates as a full two phase commit transaction managed by the Distributed Transaction Coordinator service. If the transaction proceeds without error, both ExactlyOnceReceived is incremented in the database and the AMQP message is consumed from the broker. Otherwise, ExactlyOnceReceived is unchanged and AMQP message is returned to its queue on the broker.

For the client code a few changes are made to the non-transacted example. For "exactly once" semantics, we set the AMQP "Durable" message property and enclose the transacted activities in a TransactionScope:

```
AmqpProperties myDefaults = new AmqpProperties();
myDefaults.Durable = true;
amqpBinding.DefaultMessageProperties = myDefaults;
ChannelFactory<IHelloService> channelFactory =
new ChannelFactory<IHelloService>(amqpBinding, clientEndpoint);
IHelloService clientProxy = channelFactory.CreateChannel();

using (TransactionScope ts = new TransactionScope())
{
   AmqpProperties amqpProperties = new AmqpProperties();
   clientProxy.SayHello("Greetings from WCF client");
   // increment ExactlyOnceSent counter on DB
   ts.Complete();
}
```