Apache Qpid

Open Source AMQP Messaging

Apache Qpid: Open Source AMQP Messaging

Table of Contents

List of Tables

List of Examples

Part I. Basics

Table of Contents

Chapter 1. Apache Qpid: Open Source AMQP Messaging

Enterprise Messaging systems let programs communicate by exchanging messages, much as people communicate by exchanging email. Unlike email, enterprise messaging systems provide guaranteed delivery, speed, security, and freedom from spam. Until recently, there was no open standard for Enterprise Messaging systems, so programmers either wrote their own, or used expensive proprietary systems.

AMQP Advanced Message Queuing Protocol is the first open standard for Enterprise Messaging. It is designed to support messaging for just about any distributed or business application. Routing can be configured flexibly, easily supporting common messaging paradigms like point-to-point, fanout, publish-subscribe, and request-response.

Apache Qpid implements the latest AMQP specification, providing transaction management, queuing, distribution, security, management, clustering, federation and heterogeneous multi-platform support and a lot more. And Apache Qpid is extremely fast. Apache Qpid aims to be 100% AMQP Compliant [### FIX ME ###].

1. AMQP Messaging Brokers

Qpid provides two AMQP messaging brokers:

- Implemented in C++ - high performance, low latency, and RDMA support.
- Implemented in Java - Fully JMS compliant, runs on any Java platform.

Both AMQP messaging brokers support clients in multiple languages, as long as the messaging client and the messaging broker use the same version of AMQP. See Download [### FIX ME ###] to see which messaging clients work with each broker.

2. AMQP Client APIs: C++, Java, JMS, Ruby, Python, and C#

Qpid provides AMQP Client APIs for the following languages:

- C++
- Java, fully conformant with JMS 1.1
- C# .NET, 0-10 using WCF
- Ruby
- Python

3. Operating Systems and Platforms:

The Qpid C++ broker runs on the following operating systems:

- Linux systems

- Windows
- Solaris (coming soon)

The Qpid Java broker runs on:

- Any Java platform

Qpid clients can be run on the following operating systems and platforms:

- Java:
  - any platform, production proven on Windows, Linux, Solaris
- C++:
  - Linux
  - Windows
  - Solaris (coming soon)
- C#
  - .NET

Chapter 2. AMQP (Advanced Message Queueing Protocol

AMQP Advanced Message Queuing Protocol [http://www.amqp.org/] is an open standard designed to support reliable, high-performance messaging over the Internet. AMQP can be used for any distributed or business application, and supports common messaging paradigms like point-to-point, fanout, publish-subscribe, and request-response.

Apache Qpid implements AMQP, including transaction management, queuing, clustering, federation, security, management and multi-platform support.

Apache Qpid implements the latest AMQP specification, providing transaction management, queuing, distribution, security, management, clustering, federation and heterogeneous multi-platform support and a lot more.

Apache Qpid is highly optimized, and aims to be 100% AMQP Compliant [amqp-compatibility.html].

1. Download the AMQP Specifications

AMQP version 0-10

- AMQP 0-10 Specification (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1]
- AMQP 0-10 Protocol Definition XML [https://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.xml?version=1]
- AMQP 0-10 Protocol Definition DTD [https://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.dfd?version=1]

AMQP version 0-9-1

- AMQP 0-9-1 Specification (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9-1.pdf?version=1]
- AMQP 0-9-1 Protocol Documentation (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9-1.xml?version=1]
- AMQP 0-9-1 Protocol Definitions (XML) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9-1.dtd?version=1]

AMQP version 0-9

- AMQP 0-9 Specification (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9.pdf?version=1]
- AMQP 0-9 Protocol Documentation (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9.xml?version=1]
- AMQP 0-9 Protocol Definitions (XML) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-9.dtd?version=1]

AMQP version 0-8

- AMQP 0-8 Specification (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-8.pdf?version=1]
- AMQP 0-8 Protocol Documentation (PDF) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-8.dtd?version=1]
- AMQP 0-8 Protocol Definitions (XML) [https://jira.amqp.org/confluence/download/attachments/720900/amqp0-8.xml?version=1]

Chapter 3. Getting Started

To get started with Apache Qpid, follow the steps below.

1. Download Apache Qpid.
2. Start a broker.
   - ???
   - Section 1, “ Running a Qpid C++ Broker ”
   - Chapter 6, *Managing the AMQP Messaging Broker* (AMQP 0-10, works with the Qpid C++ broker)
3. Run an example program from the downloaded software, or from the following URLs (these are svn URLs, which you can use to browse the examples or check them out):
   - C++ (AMQP 0-10):
     - Examples:

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/cpp/examples/
     - Running the C++ Examples:

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/cpp/examples/README.txt
   - Java JMS (AMQP 0-10):
     - Examples:

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/
     - Script for Running the Java JMS Examples

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/runSample.sh
   - Python (AMQP 0-10):
     - Examples:

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/python/examples/
     - Running the Python Examples

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/python/examples/README
   - Ruby (AMQP 0-10):
     - Examples:

       https://svn.apache.org/repos/asf/qpid/trunk/qpid/ruby/examples/
   - .NET (AMQP 0-10):
     - Examples:

http://svn.apache.org/viewvc/qpid/trunk/qpid/dotnet/client-010/examples/

- Section 1.1.1, “ Tutorial ”

4. Read the API Guides and Documentation
   - C++ Client API (AMQP 0-10)

     http://qpid.apache.org/docs/api/cpp/html/index.html
   - ???
   - Python Client API (AMQP 0-10)

     http://qpid.apache.org/docs/api/python/html/index.html
5. Get your Questions Answered
   - Read the ???
   - Ask a question on the user list

     mailto:users-subscribe@qpid.apache.org

Chapter 4. Download Apache Qpid

1. Production Releases

These releases are well tested and appropriate for production use. 0.5 is the latest release of Qpid.

Qpid supports the latest version of AMQP 0-10, and some components also the AMQP 0-8 and 0-9, earlier versions. The Java Broker and Client provide protocol negotiation. Other versions can be found at http://www.apache.org/dist/qpid/

For details on cross component compatibility among releases, see: AMQP Release Compatibility for Qpid| AMQP compatibility [### FIX ME ###]

If you have any questions about these releases, please mail the user list (qpid-user@incubator.apache.org [mailto:qpid-user@incubator.apache.org]).

2. 0.5 Release

2.1. Multiple Component Packages

Table 4.1.

Component	Download	AMQP 0-10	AMQP 0-8/0-9
Full release & keys	http://www.apache.org/dist/qpid/0.5/	Y	Y
C++ broker & client	http://www.apache.org/dist/qpid/0.5/qpid-cpp-0.5.tar.gz	Y	
Java broker, client & tools	http://www.apache.org/dist/qpid/0.5/qpid-java-0.5.tar.gz	client	Y

2.2. Single Component Package

Table 4.2. Broker

Language	Download	AMQP 0-10	AMQP 0-8/0-9
Java	http://www.apache.org/dist/qpid/0.5/qpid-java-broker-0.5.tar.gz		Y

Table 4.3. Client

Language	Download	AMQP 0-10	AMQP 0-8/0-9
C# (.NET, WCF, Excel) 0-10 client (C++ Broker Compatible)	http://www.apache.org/dist/qpid/0.5/qpid-dotnet-0-10-0.5.zip	Y	

Language	Download	AMQP 0-10	AMQP 0-8/0-9
C# (.NET) 0-8 client (Java Broker Compatible)	http://www.apache.org/dist/qpid/0.5/qpid-dotnet-0-8-0.5.zip		Y
Java	http://www.apache.org/dist/qpid/0.5/qpid-java-client-0.5.tar.gz	Y	Y
Python	http://www.apache.org/dist/qpid/0.5/qpid-python-0.5.tar.gz	Y	Y
Ruby	http://www.apache.org/dist/qpid/0.5/qpid-ruby-0.5.tar.gz	Y	Y

Table 4.4. C++ broker management

Component	Download	AMQP 0-10
cmd line (packaged with python)	http://www.apache.org/dist/qpid/0.5/qpid-python-0.5.tar.gz	Y
QMan JMX bridge, WS-DM	http://www.apache.org/dist/qpid/0.5/qpid-management-client-0.5.tar.gz	Y

Table 4.5. Java broker management

Component	Download
Eclipse RCP client	Linux x86 [http://www.apache.org/dist/qpid/0.5/qpid-management-eclipse-plugin-0.5-linux-gtk-x86.tar.gz] Linux x86_64 [http://www.apache.org/dist/qpid/0.5/qpid-management-eclipse-plugin-0.5-linux-gtk-x86_64.tar.gz] Mac OS X [http://www.apache.org/dist/qpid/0.5/qpid-management-eclipse-plugin-0.5-macosx.zip] Windows x86 [http://www.apache.org/dist/qpid/0.5/qpid-management-eclipse-plugin-0.5-win32-win32-x86.zip]
Command line interface	http://www.apache.org/dist/qpid/0.5/qpid-management-tools-qpid-cli-0.5.tar.gz

3. QpidComponents.org

http://QpidComponents.org provides further components for Apache Qpid, including both persistence and management tools. These components are open source, but are not developed as part of the Apache Qpid project due to licensing or other restrictions.

4. Contributed C++ Packages

4.1. Pre-built Linux Packages

4.1.1. Fedora 8, 9, 10

On Fedora, Qpid can be installed using yum. Because Java RPMs are not yet available in Fedora repos, the Java client is not in these distributions.

To install the server:

```
# yum install qpidd
```

To install C++ and Python clients:

```
# yum install qpidc-devel
```

```
# yum install amqp python-qpid
```

To install documentation:

```
# yum install rhm-docs
```

To install persistence using an external store module:

```
# yum install rhm
```

4.2. Windows Installer

The Windows installer is available from http://www.apache.org/dist/qpid/0.5-windows/qpidc-0.5.msi. It is built from the 0.5 C++ broker and client source distribution listed above. It has been tested for Windows XP SP2 and above.

The Windows executables require the Visual C++ 2008 SP1 run-time components. If the Visual C++ 2008 SP1 runtime is not available, the Qpid broker will not execute. If you intend to run the broker and Visual C++ 2008 is not installed, you must install the Visual C++ 2008 SP1 Redistributable. Please see http://www.microsoft.com/downloads/details.aspx?familyid=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2&displaylang=en for download and installation instructions.

If you intend to develop Qpid client applications using this kit, you should install Boost version 1.35 [http://www.boostpro.com/download/boost_1_35_0_setup.exe] (please be sure to select VC9 support when installing) in addition to Visual Studio 2008 SP1.

5. Source Code Repository

The latest version of the code is always available in the Source Repository.

Part II. AMQP Messaging Broker (Implemented in C++)

Qpid provides two AMQP messaging brokers:

- Implemented in C++ - high performance, low latency, and RDMA support.
- Implemented in Java - Fully JMS compliant, runs on any Java platform.

Both AMQP messaging brokers support clients in multiple languages, as long as the messaging client and the messaging broker use the same version of AMQP. See ??? to see which messaging clients work with each broker.

This section contains information specific to the broker that is implemented in C++.

Chapter 5. Running the AMQP Messaging Broker

1. Running a Qpid C++ Broker

1.1. Building the C++ Broker and Client Libraries

The root directory for the C++ distribution is named qpidc-0.4. The README file in that directory gives instructions for building the broker and client libraries. In most cases you will do the following:

```
[qpidc-0.4]$ ./configure
[qpidc-0.4]$ make
```

1.2. Running the C++ Broker

Once you have built the broker and client libraries, you can start the broker from the command line:

```
[qpidc-0.4]$ src/qpidd
```

Use the --daemon option to run the broker as a daemon process:

```
[qpidc-0.4]$ src/qpidd --daemon
```

You can stop a running daemon with the --quit option:

```
[qpidc-0.4]$ src/qpidd --quit
```

You can see all available options with the --help option

```
[qpidc-0.4]$ src/qpidd --help
```

1.3. Most common questions getting qpidd running

1.3.1. Error when starting broker: "no data directory"

The qpidd broker requires you to set a data directory or specify --no-data-dir (see help for more details). The data directory is used for the journal, so it is important when reliability counts. Make sure your process has write permission to the data directory.

The default location is

```
/lib/var/qpidd
```

An alternate location can be set with --data-dir

1.3.2. Error when starting broker: "that process is locked"

Note that when qpidd starts it creates a lock file is data directory are being used. If you have a un-controlled exit, please mail the trace from the core to the dev@qpid.apache.org mailing list. To clear the lock run

```
./qpidd -q
```

It should also be noted that multiple brokers can be run on the same host. To do so set alternate data directories for each qpidd instance.

1.3.3. Using a configuration file

Each option that can be specified on the command line can also be specified in a configuration file. To see available options, use --help on the command line:

```
./qpidd --help
```

A configuration file uses name/value pairs, one on each line. To convert a command line option to a configuration file entry:

a.) remove the '--' from the beginning of the option. b.) place a '=' between the option and the value (use *yes* or *true* to enable options that take no value when specified on the command line). c.) place one option per line.

For instance, the --daemon option takes no value, the --log-to-syslog option takes the values yes or no. The following configuration file sets these two options:

```
daemon=yes
log-to-syslog=yes
```

1.3.4. Can I use any Language client with the C++ Broker?

Yes, all the clients work with the C++ broker; it is written in C+, *but uses the AMQP wire protocol. Any broker can be used with any client that uses the same AMQP version. When running the C+* broker, it is highly recommended to run AMQP 0-10.

Note that JMS also works with the C++ broker.

1.4. Authentication

1.4.1. Linux

The PLAIN authentication is done on a username+password, which is stored in the sasldb_path file. Usernames and passwords can be added to the file using the command:

```
saslpasswd2 -f /var/lib/qpidd/qpidd.sasldb -u <REALM> <USER>
```

The REALM is important and should be the same as the --auth-realm option to the broker. This lets the broker properly find the user in the sasldb file.

Existing user accounts may be listed with:

```
sasldblistusers2 -f /var/lib/qpidd/qpidd.sasldb
```

NOTE: The sasldb file must be readable by the user running the qpidd daemon, and should be readable only by that user.

1.4.2. Windows

On Windows, the users are authenticated against the local machine. You should add the appropriate users using the standard Windows tools (Control Panel->User Accounts). To run many of the examples, you will need to create a user "guest" with password "guest".

If you cannot or do not want to create new users, you can run without authentication by specifying the no-auth option to the broker.

1.5. Slightly more complex configuration

The easiest way to get a full listing of the broker's options are to use the --help command, run it locally for the latest set of options. These options can then be set in the conf file for convenience (see above)

```
./qpidd --help

Usage: qpidd OPTIONS
Options:
  -h [ --help ]                    Displays the help message
  -v [ --version ]                 Displays version information
  --config FILE (/etc/qpidd.conf)  Reads configuration from FILE

Module options:
  --module-dir DIR (/usr/lib/qpidd)  Load all .so modules in this directory
  --load-module FILE                 Specifies additional module(s) to be loaded
  --no-module-dir                    Don't load modules from module directory

Broker Options:
  --data-dir DIR (/var/lib/qpidd)   Directory to contain persistent data generated
  --no-data-dir                     Don't use a data directory.  No persistent
                                    configuration will be loaded or stored
  -p [ --port ] PORT (5672)         Tells the broker to listen on PORT
  --worker-threads N (3)            Sets the broker thread pool size
  --max-connections N (500)         Sets the maximum allowed connections
  --connection-backlog N (10)       Sets the connection backlog limit for the
                                    server socket
  --staging-threshold N (5000000)   Stages messages over N bytes to disk
  -m [ --mgmt-enable ] yes|no (1)   Enable Management
  --mgmt-pub-interval SECONDS (10)  Management Publish Interval
  --ack N (0)                       Send session.ack/solicit-ack at least every
                                    N frames. 0 disables voluntary ack/solitict
                                   -ack

Daemon options:
  -d [ --daemon ]             Run as a daemon.
  -w [ --wait ] SECONDS (10)  Sets the maximum wait time to initialize the
                              daemon. If the daemon fails to initialize, prints
                              an error and returns 1
```

```
  -c [ --check ]              Prints the daemon's process ID to stdout and
                              returns 0 if the daemon is running, otherwise
                              returns 1
  -q [ --quit ]               Tells the daemon to shut down
Logging options:
  --log-output FILE (stderr)  Send log output to FILE. FILE can be a file name
                              or one of the special values:
                              stderr, stdout, syslog
  -t [ --trace ]              Enables all logging
  --log-enable RULE (error+)  Enables logging for selected levels and component
                              s. RULE is in the form 'LEVEL+:PATTERN'
                              Levels are one of:
                              trace debug info notice warning error critical
                              For example:
                              '--log-enable warning+' logs all warning, error
                              and critical messages.
                              '--log-enable debug:framing' logs debug messages
                              from the framing namespace. This option can be
                              used multiple times
  --log-time yes|no (1)       Include time in log messages
  --log-level yes|no (1)      Include severity level in log messages
  --log-source yes|no (0)     Include source file:line in log messages
  --log-thread yes|no (0)     Include thread ID in log messages
  --log-function yes|no (0)   Include function signature in log messages
```

1.6. Loading extra modules

By default the broker will load all the modules in the module directory, however it will NOT display options for modules that are not loaded. So to see the options for extra modules loaded you need to load the module and then add the help command like this:

```
./qpidd --load-module libbdbstore.so --help
Usage: qpidd OPTIONS
Options:
  -h [ --help ]                   Displays the help message
  -v [ --version ]                Displays version information
  --config FILE (/etc/qpidd.conf)  Reads configuration from FILE


 / .... non module options would be here ... /


Store Options:
  --store-directory DIR     Store directory location for persistence (overrides
                            --data-dir)
  --store-async yes|no (1)  Use async persistence storage - if store supports
                            it, enables AIO O_DIRECT.
  --store-force yes|no (0)  Force changing modes of store, will delete all
                            existing data if mode is changed. Be SURE you want
                            to do this!
  --num-jfiles N (8)        Number of files in persistence journal
  --jfile-size-pgs N (24)   Size of each journal file in multiples of read
                            pages (1 read page = 64kiB)
```

2. Cheat Sheet for configuring Queue Options

2.1. Configuring Queue Options

The C++ Broker M4 or later supports the following additional Queue constraints.

2.1.1. Applying Queue Sizing Constraints

This allows to specify how to size a queue and what to do when the sizing constraints have been reached. The queue size can be limited by the number messages (message depth) or byte depth on the queue.

Once the Queue meets/ exceeds these constraints the follow policies can be applied

- REJECT - Reject the published message
- FLOW_TO_DISK - Flow the messages to disk, to preserve memory
- RING - start overwriting messages in a ring based on sizing. If head meets tail, advance head
- RING_STRICT - start overwriting messages in a ring based on sizing. If head meets tail, AND the consumer has the tail message acquired it will reject

Examples:

Create a queue an auto delete queue that will support 100 000 bytes, and then REJECT

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

    QueueOptions qo;
    qo.setSizePolicy(REJECT,100000,0);

    session.queueDeclare(arg::queue=queue, arg::autoDelete=true, arg::arguments=qo
```

Create a queue that will support 1000 messages into a RING buffer

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

    QueueOptions qo;
    qo.setSizePolicy(RING,0,1000);
```

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

2.1.2. Changing the Queue ordering Behaviors (FIFO/LVQ)

The default ordering in a queue in Qpid is FIFO. However additional ordering semantics can be used namely LVQ (Last Value Queue). Last Value Queue is define as follows.

If I publish symbols RHT, IBM, JAVA, MSFT, and then publish RHT before the consumer is able to consume RHT, that message will be over written in the queue and the consumer will receive the last published value for RHT.

Example:

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

    QueueOptions qo;
    qo.setOrdering(LVQ);

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

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

    ....
    for each message, set the into application headers before transfer
    message.getHeaders().setString(key,"RHT");
```

Notes:

- Messages that are dequeued and the re-queued will have the following exceptions. a.) if a new message has been queued with the same key, the re-queue from the consumer, will combine these two messages. b.) If an update happens for a message of the same key, after the re-queue, it will not update the re-queued message. This is done to protect a client from being able to adversely manipulate the queue.
- Acquire: When a message is acquired from the queue, no matter it's position, it will behave the same as a dequeue
- LVQ does not support durable messages. If the queue or messages are declared durable on an LVQ, the durability will be ignored.

A fully worked Section 6.4, “ LVQ Program Example ” can be found here

2.1.3. Setting additional behaviors

2.1.3.1. Persist Last Node

This option is used in conjunction with clustering. It allows for a queue configured with this option to persist transient messages if the cluster fails down to the last node. If additional nodes in the cluster are restored it will stop persisting transient messages.

Note

- if a cluster is started with only one active node, this mode will not be triggered. It is only triggered the first time the cluster fails down to 1 node.

- The queue MUST be configured durable

Example:

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

    QueueOptions qo;
    qo.clearPersistLastNode();

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

2.1.3.2. Queue event generation

This option is used to determine whether enqueue/dequeue events representing changes made to queue state are generated. These events can then be processed by plugins such as that used for Section 7, “ Queue State Replication ”.

Example:

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

    QueueOptions options;
    options.enableQueueEvents(1);
    session.queueDeclare(arg::queue="my-queue", arg::arguments=options);
```

The boolean option indicates whether only enqueue events should be generated. The key set by this is 'qpid.queue_event_generation' and the value is and integer value of 1 (to replicate only enqueue events) or 2 (to replicate both enqueue and dequeue events).

2.1.4. Other Clients

Note that these options can be set from any client. QueueOptions just correctly formats the arguments passed to the QueueDeclare() method.

3. Cheat Sheet for configuring Exchange Options

3.1. Configuring Exchange Options

The C++ Broker M4 or later supports the following additional Exchange options in addition to the standard AMQP define options

- Exchange Level Message sequencing
- Initial Value Exchange

Note that these features can be used on any exchange type, that has been declared with the options set.

It also supports an additional option to the bind operation on a direct exchange

- Exclusive binding for key

3.1.1. Exchange Level Message sequencing

This feature can be used to place a sequence number into each message's headers, based on the order they pass through an exchange. The sequencing starts at 0 and then wraps in an AMQP int64 type.

The field name used is "qpid.msg_sequence"

To use this feature an exchange needs to be declared specifying this option in the declare

```
....
    FieldTable args;
    args.setInt("qpid.msg_sequence",1);

...
    // now declare the exchange
    session.exchangeDeclare(arg::exchange="direct", arg::arguments=args);
```

Then each message passing through that exchange will be numbers in the application headers.

```
    unit64_t seqNo;
    //after message transfer
    seqNo = message.getHeaders().getAsInt64("qpid.msg_sequence");
```

3.1.2. Initial Value Exchange

This feature caches a last message sent to an exchange. When a new binding is created onto the exchange it will then attempt to route this cached messaged to the queue, based on the binding. This allows for topics or the creation of configurations where a new consumer can receive the last message sent to the broker, with matching routing.

To use this feature an exchange needs to be declared specifying this option in the declare

```
....
    FieldTable args;
    args.setInt("qpid.ive",1);

...
    // now declare the exchange
    session.exchangeDeclare(arg::exchange="direct", arg::arguments=args);
```

now use the exchange in the same way you would use any other exchange.

3.1.3. Exclusive binding for key

Direct exchanges in qpidd support a qpid.exclusive-binding option on the bind operation that causes the binding specified to be the only one for the given key. I.e. if there is already a binding at this exchange with this key it will be atomically updated to bind the new queue. This means that the binding can be changed concurrently with an incoming stream of messages and each message will be routed to exactly one queue.

```
....
    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);

...
```

4. Using Broker Federation

4.1. Introduction

Please note: Whereas broker federation was introduced in the M3 milestone release, the discussion in this document is based on the richer capabilities of federation in the M4 release.

4.2. What Is Broker Federation?

The Qpid C++ messaging broker supports broker federation, a mechanism by which large messaging networks can be built using multiple brokers. Some scenarios in which federation is useful:

- *Connecting disparate locations across a wide area network.* In this case full connectivity across the enterprise can be achieved while keeping local message traffic isolated to a single location.
- *Departmental brokers* that have a policy which controls the flow of inter-departmental message traffic.
- *Scaling of capacity* for expensive broker operations. High-function exchanges like the XML exchange can be replicated to scale performance.
- *Co-Resident brokers* Some applications benefit from having a broker co-resident with the client. This is particularly true if the client produces data that must be delivered reliably but connectivity to the consumer(s) is non-reliable. In this case, a co-resident broker provides queueing and durablilty not available in the client alone.
- *Bridging disjoint IP networks.* Message brokers can be configured to allow message connectivity between networks where there is no IP connectivity. For example, an isolated, private IP network can have messaging connectivity to brokers in other outside IP networks.

4.3. The qpid-route Utility

The qpid-route command line utility is provided with the Qpid broker. This utility is used to configure federated networks of brokers and to view the status and topology of networks.

qpid-route accesses the managed brokers remotely. It does not need to be invoked from the same host on which the broker is running. If network connectivity permits, an entire enterprise can be configured from a single location.

In the following sections, federation concepts will be introduced and illustrated using qpid-route.

4.3.1. Links and Routes

Federation occurs when a *link* is established between two brokers and one or more *routes* are created within that link. A *link* is a transport level connection (tcp, rdma, ssl, etc.) initiated by one broker and accepted by another. The initiating broker assumes the role of *client* with regard to the connection. The accepting broker annotates the connection as being for federation but otherwise treats it as a normal client connection.

A *route* is associated with an AMQP session established over the link connection. There may be multiple routes sharing the same link. A route controls the flow of messages across the link between brokers. Routes always consist of a session and a subscription for consuming messages. Depending on the configuration, a route may have a private queue on the source broker with a binding to an exchange on that broker.

Routes are unidirectional. A single route provides for the flow of messages in one direction across a link. If bidirectional connectivity is required (and it almost always is), then a pair of routes must be created, one for each direction of message flow.

The qpid-route utility allows the administrator to configure and manage links and routes separately. However, when a route is created and a link does not already exist, qpid-route will automatically create the link. It is typically not necessary to create a link by itself. It is, however, useful to get a list of links and their connection status from a broker:

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

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

The example above shows a *link list* query to the broker at "localhost:10001". In the example, this broker has three links to other brokers. Two are operational and the third is waiting to connect because there is not currently a broker listening at that address.

4.3.1.1. The Life Cycle of a Link

When a link is created on a broker, that broker attempts to establish a transport-level connection to the peer broker. If it fails to connect, it retries the connection at an increasing time interval. If the connection fails due to authentication failure, it will not continue to retry as administrative intervention is needed to fix the problem.

If an operational link is disconnected, the initiating broker will attempt to re-establish the connection with the same interval back-off.

The shortest retry-interval is 2 seconds and the longest is 64 seconds. Once enough consecutive retries have occurred that the interval has grown to 64 seconds, the interval will then stay at 64 seconds.

4.3.1.2. Durable Links and Routes

If, when a link or a route is created using qpid-route, the --durable option is used, it shall be durable. This means that its life cycle shall span restarts of the broker. If the broker is shut down, when it is restarted, the link will be restored and will begin establishing connectivity.

A non-durable route can be created for a durable link but a durable route cannot be created for a non-durable link.

```
$ qpid-route dynamic add localhost:10003 localhost:10004 fed.topic
$ qpid-route dynamic add localhost:10003 localhost:10004 fed.topic2 --durable
Failed: Can't create a durable route on a non-durable link
```

In the above example, a transient (non-durable) dynamic route was created between localhost:10003 and localhost:10004. Because there was no link in place, a new transient link was created. The second command is attempting to create a durable route over the same link and is rejected as illegal.

4.3.2. Dynamic Routing

Dynamic routing provides the simplest configuration for a network of brokers. When configuring dynamic routing, the administrator need only express the logical topology of the network (i.e. which pairs of brokers are connected by a unidirectional route). Queue configuration and bindings are handled automatically by the brokers in the network.

Dynamic routing uses the *Distributed Exchange* concept. From the client's point of view, all of the brokers in the network collectively offer a single logical exchange that behaves the same as a single exchange in a single broker. Each client connects to its local broker and can bind its queues to the distributed exchange and publish messages to the exchange.

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

When creating a dynamic routing network, The type and name of the exchange must be the same on each broker. It is *strongly* recommended that dynamic routes *NOT* be created using the standard exchanges (that is unless all messaging is intended to be federated).

A simple, two-broker network can be configured by creating an exchange on each broker then a pair of dynamic routes (one for each direction of message flow):

Create exchanges:

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

Create dynamic routes:

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

Information about existing routes can be gotten by querying each broker individually:

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

A nicer way to view the topology is to use *qpid-route route map*. The argument to this command is a single broker that serves as an entry point. *qpid-route* will attempt to recursively find all of the brokers involved in federation relationships with the starting broker and map all of the routes it finds.

```
$ qpid-route route map localhost:10003

Finding Linked Brokers:
    localhost:10003... Ok
    localhost:10004... Ok
```

```
Dynamic Routes:

  Exchange fed.topic:
    localhost:10004 <=> localhost:10003

Static Routes:
  none found
```

More extensive and realistic examples are supplied later in this document.

4.3.3. Static Routing

Dynamic routing provides simple, efficient, and automatic handling of the bindings that control routing as long as the configuration keeps within a set of constraints (i.e. exchanges of the same type and name, bidirectional traffic flow, etc.). However, there are scenarios where it is useful for the administrator to have a bit more control over the details. In these cases, static routing is appropriate.

4.3.3.1. Exchange Routes

An exchange route is like a dynamic route except that the exchange binding is statically set at creation time instead of dynamically tracking changes in the network.

When an exchange route is created, a private queue (auto-delete, exclusive) is declared on the source broker. The queue is bound to the indicated exchange with the indicated key and the destination broker subscribes to the queue with a destination of the indicated exchange. Since only one exchange name is supplied, this means that exchange routes require that the source and destination exchanges have the same name.

Static exchange routes are added and deleted using *qpid-route route add* and *qpid-route route del* respectively. The following example creates a static exchange route with a binding key of "global.#" on the default topic exchange:

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

The route can be viewed by querying the originating broker (the destination in this case, see discussion of push and pull routes for more on this):

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

Alternatively, the *route map* feature can be used to view the topology:

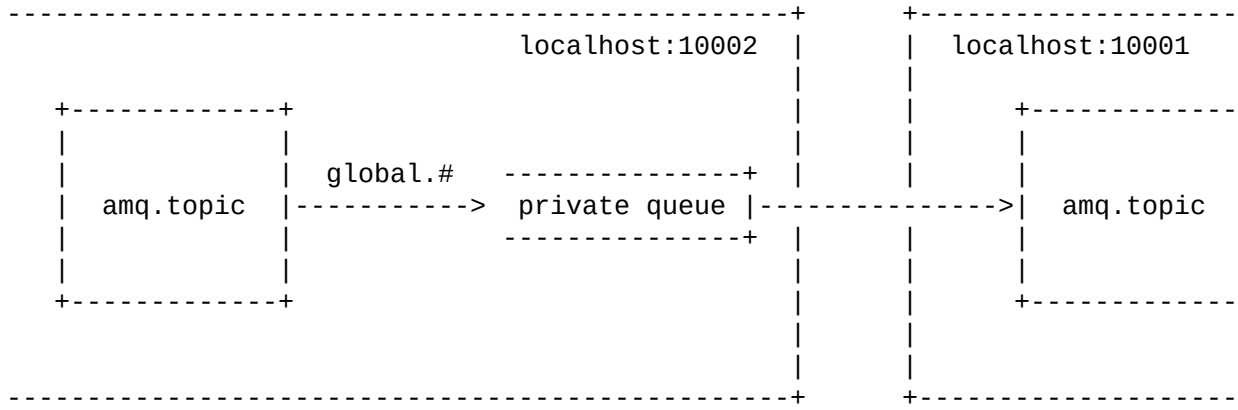
```
$ qpid-route route map localhost:10001

Finding Linked Brokers:
    localhost:10001... Ok
    localhost:10002... Ok

Dynamic Routes:
  none found

Static Routes:
```

```
localhost:10001(ex=amq.topic) <= localhost:10002(ex=amq.topic) key=global.#
```

This example causes messages delivered to the *amq.topic* exchange on broker *localhost:10002* that have a key that matches *global.#* (i.e. starts with the string "global.") to be delivered to the *amq.topic* exchange on broker *localhost:10001*. This delivery will occur regardless of whether there are any consumers on *localhost:10001* that will receive the messages.

Note that this is a uni-directional route. No messages will be forwarded in the opposite direction unless another static route is created in the other direction.

The following diagram illustrates the result, in terms of AMQP objects, of the example static exchange route. In this diagram, the exchanges, both named "amq.topic" exist prior to the creation of the route. The creation of the route causes the private queue, the binding, and the subscription of the queue to the destination to be created.



4.3.3.2. Queue Routes

A queue route causes the destination broker to create a subscription to a pre-existing, possibly shared, queue on the source broker. There's no requirement that the queue be bound to any particular exchange. Queue routes can be used to connect exchanges of different names and/or types. They can also be used to distribute or balance traffic across multiple destination brokers.

Queue routes are created and deleted using the *qpid-route queue add* and *qpid-route queue del* commands respectively. The following example creates a static queue route to a public queue called "public" that feeds the amq.fanout exchange on the destination:

Create a queue on the source broker:

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

Create a queue route to the new queue

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

4.3.3.3. Pull vs. Push Routes

When qpid-route creates or deletes a route, it establishes a connection to one of the brokers involved in the route and configures that broker. The configured broker then takes it upon itself to contact the other broker and exchange whatever information is needed to complete the setup of the route.

The notion of *push* vs. *pull* is concerned with whether the configured broker is the source or the destination. The normal case is the pull route, where qpid-route configures the destination to pull messages from the source. A push route occurs when qpid-route configures the source to push messages to the destination.

Dynamic routes are always pull routes. Static routes are normally pull routes but may be inverted by using the src-local option when creating (or deleting) a route. If src-local is specified, qpid-route will make its connection to the source broker rather than the destination and configure the route to push rather than pull.

Push routes are useful in applications where brokers are co-resident with data sources and are configured to send data to a central broker. Rather than configure the central broker for each source, the sources can be configured to send to the destination.

4.3.4. qpid-route Summary and Options

```
$ qpid-route
Usage:  qpid-route [OPTIONS] dynamic add <dest-broker> <src-broker> <exchange> [ta
        qpid-route [OPTIONS] dynamic del <dest-broker> <src-broker> <exchange>

        qpid-route [OPTIONS] route add   <dest-broker> <src-broker> <exchange> <ro
        qpid-route [OPTIONS] route del   <dest-broker> <src-broker> <exchange> <ro
        qpid-route [OPTIONS] queue add   <dest-broker> <src-broker> <exchange> <qu
        qpid-route [OPTIONS] queue del   <dest-broker> <src-broker> <exchange> <qu
        qpid-route [OPTIONS] route list  [<dest-broker>]
        qpid-route [OPTIONS] route flush [<dest-broker>]
        qpid-route [OPTIONS] route map   [<broker>]

        qpid-route [OPTIONS] link add  <dest-broker> <src-broker>
        qpid-route [OPTIONS] link del  <dest-broker> <src-broker>
        qpid-route [OPTIONS] link list [<dest-broker>]

Options:
    --timeout seconds (10)   Maximum time to wait for broker connection
    -v [ --verbose ]         Verbose output
    -q [ --quiet ]           Quiet output, don't print duplicate warnings
    -d [ --durable ]         Added configuration shall be durable
    -e [ --del-empty-link ]  Delete link after deleting last route on the link
    -s [ --src-local ]       Make connection to source broker (push route)
    --ack N                  Acknowledge transfers over the bridge in batches of N
    -t <transport> [ --transport <transport>]
                             Specify transport to use for links, defaults to tcp

  dest-broker and src-broker are in the form:  [username/password@] hostname | ip-
  ex:  localhost, 10.1.1.7:10000, broker-host:10000, guest/guest@localhost
```

There are several transport options available for the federation link:

Table 5.1. Transport Options for Federation

Transport	Description
tcp	(default) A cleartext TCP connection
ssl	A secure TLS/SSL over TCP connection
rdma	A Connection using the RDMA interface (typically for an Infiniband network)

The *tag* and *exclude-list* arguments are not needed. They have been left in place for backward compatibility and for advanced users who might have very unusual requirements. If you're not sure if you need them, you don't. Leave them alone. If you must know, please refer to "Message Loop Prevention" in the advanced topics section below. The prevention of message looping is now automatic and requires no user action.

If the link between the two sites has network latency, this can be compensated for by increasing the ack frequency with --ack N to achieve better batching across the link between the two sites.

4.3.5. Caveats, Limitations, and Things to Avoid

4.3.5.1. Redundant Paths

The current implementation of federation in the M4 broker imposes constraints on redundancy in the topology. If there are parallel paths from a producer to a consumer, multiple copies of messages may be received.

A future release of Qpid will solve this problem by allowing redundant paths with cost metrics. This will allow the deployment of networks that are tolerant of connection or broker loss.

4.3.5.2. Lack of Flow Control

M4 broker federation uses unlimited flow control on the federation sessions. Flow control back-pressure will not be applied on inter-broker subscriptions.

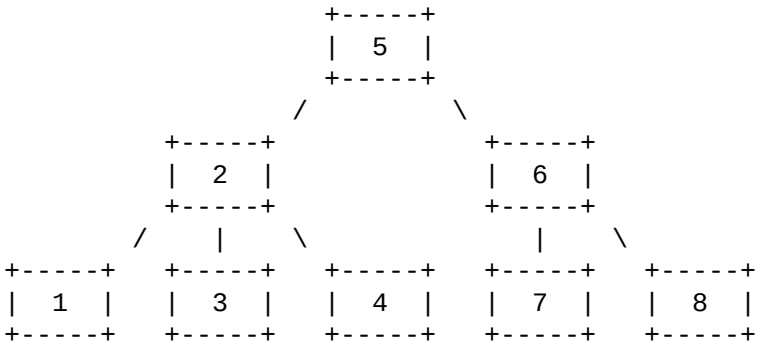
4.3.5.3. Lack of Cluster Failover Support

The client functionality embedded in the broker for inter-broker links does not currently support cluster fail-over. This will be added in a subsequent release.

4.4. Example Scenarios

4.4.1. Using QPID to bridge disjoint IP networks

4.4.1.1. Multi-tiered topology



This topology can be configured using the following script.

```
##
## Define URLs for the brokers
##
broker1=localhost:10001
```

```
broker2=localhost:10002
broker3=localhost:10003
broker4=localhost:10004
broker5=localhost:10005
broker6=localhost:10006
broker7=localhost:10007
broker8=localhost:10008

##
## Create Topic Exchanges
##
qpid-config -a $broker1 add exchange topic fed.topic
qpid-config -a $broker2 add exchange topic fed.topic
qpid-config -a $broker3 add exchange topic fed.topic
qpid-config -a $broker4 add exchange topic fed.topic
qpid-config -a $broker5 add exchange topic fed.topic
qpid-config -a $broker6 add exchange topic fed.topic
qpid-config -a $broker7 add exchange topic fed.topic
qpid-config -a $broker8 add exchange topic fed.topic

##
## Create Topic Routes
##
qpid-route dynamic add $broker1 $broker2 fed.topic
qpid-route dynamic add $broker2 $broker1 fed.topic

qpid-route dynamic add $broker3 $broker2 fed.topic
qpid-route dynamic add $broker2 $broker3 fed.topic

qpid-route dynamic add $broker4 $broker2 fed.topic
qpid-route dynamic add $broker2 $broker4 fed.topic

qpid-route dynamic add $broker2 $broker5 fed.topic
qpid-route dynamic add $broker5 $broker2 fed.topic

qpid-route dynamic add $broker5 $broker6 fed.topic
qpid-route dynamic add $broker6 $broker5 fed.topic

qpid-route dynamic add $broker6 $broker7 fed.topic
qpid-route dynamic add $broker7 $broker6 fed.topic

qpid-route dynamic add $broker6 $broker8 fed.topic
qpid-route dynamic add $broker8 $broker6 fed.topic
```

4.4.1.2. Load-sharing across brokers

4.5. Advanced Topics

4.5.1. Federation Queue Naming

4.5.2. Message Loop Prevention

5. SSL

5.1. SSL How to

5.1.1. C++ broker (M4 and up)

- You need to get a certificate signed by a CA, trusted by your client.
- If you require client authentication, the clients certificate needs to be signed by a CA trusted by the broker.
- Setting up the certificates for testing.
  - For testing purposes you could use the ??? to setup your certificates.
  - In summary you need to create a root CA and import it to the brokers certificate data base.
  - Create a certificate for the broker, sign it using the root CA and then import it into the brokers certificate data base.
- Load the acl module using --load-module or if loading more than one module, copy ssl.so to the location pointed by --module-dir

```
Ex if running from source. ./qpidd --load-module /libs/ssl.so
```

- Specify the password file (a plain text file with the password), certificate database and the brokers certificate name using the following options

```
Ex ./qpidd ... --ssl-cert-password-file ~/pfile --ssl-cert-db ~/server_db/ --ssl
```

- If you require client authentication you need to add --ssl-require-client-authentication as a command line argument.
- Please note that the default port for SSL connections is 5671, unless specified by --ssl-port

Here is an example of a broker instance that requires SSL client side authenticaiton

```
./qpidd ./qpidd --load-module /libs/ssl.so --ssl-cert-password-file ~/pfile --ssl-
```

5.1.2. Java Client (M4 and up)

- This guide is for connecting with the Qpid c++ broker.
- Setting up the certificates for testing. In summary,
  - You need to import the trusted CA in your trust store and keystore
  - Generate keys for the certificate in your key store
  - Create a certificate request using the generated keys
  - Create a certficate using the request, signed by the trusted CA.

- Import the signed certificate into your keystore.

- Pass the following JVM arguments to your client.

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

5.1.3. .Net Client (M4 and up)

- If the Qpid broker requires client authentication then you need to get a certificate signed by a CA, trusted by your client.

Use the connectSSL instead of the standard connect method of the client interface.

connectSSL signature is as follows:

```
public void connectSSL(String host, int port, String virtualHost, String username,
```

Where

- host: Host name on which a Qpid broker is deployed
- port: Qpid broker port
- virtualHost: Qpid virtual host name
- username: User Name
- password: Password
- serverName: Name of the SSL server
- certPath: Path to the X509 certificate to be used when the broker requires client authentication
- rejectUntrusted: If true connection will not be established if the broker is not trusted (the server certificate must be added in your truststore)

5.1.4. Python & Ruby Client (M4 and up)

Simply use amqps:// in the URL string as defined above

6. LVQ

6.1. Understanding LVQ

Last Value Queues are useful youUser Documentation are only interested in the latest value entered into a queue. LVQ semantics are typically used for things like stock symbol updates when all you care about is the latest value for example.

Qpid C++ M4 or later supports two types of LVQ semantics:

- LVQ
- LVQ_NO_BROWSE

6.2. LVQ semantics:

LVQ uses a header for a key, if the key matches it replaces the message in-place in the queue except a.) if the message with the matching key has been acquired b.) if the message with the matching key has been browsed In these two cases the message is placed into the queue in FIFO, if another message with the same key is received it will the 'un-accessed' message with the same key will be replaced

These two exceptions protect the consumer from missing the last update where a consumer or browser accesses a message and an update comes with the same key.

An example

```
[localhost tests]$ ./lvqtest --mode create_lvq
[localhost tests]$ ./lvqtest --mode write
Sending Data: key1=key1.0x7fffdf3f3180
Sending Data: key2=key2.0x7fffdf3f3180
Sending Data: key3=key3.0x7fffdf3f3180
Sending Data: key1=key1.0x7fffdf3f3180
Sending Data: last=last
[localhost tests]$ ./lvqtest --mode browse
Receiving Data:key1.0x7fffdf3f3180
Receiving Data:key2.0x7fffdf3f3180
Receiving Data:key3.0x7fffdf3f3180
Receiving Data:last
[localhost tests]$ ./lvqtest --mode write
Sending Data: key1=key1.0x7fffe4c7fa10
Sending Data: key2=key2.0x7fffe4c7fa10
Sending Data: key3=key3.0x7fffe4c7fa10
Sending Data: key1=key1.0x7fffe4c7fa10
Sending Data: last=last
[localhost tests]$ ./lvqtest --mode browse
Receiving Data:key1.0x7fffe4c7fa10
Receiving Data:key2.0x7fffe4c7fa10
Receiving Data:key3.0x7fffe4c7fa10
Receiving Data:last
[localhost tests]$ ./lvqtest --mode consume
Receiving Data:key1.0x7fffdf3f3180
Receiving Data:key2.0x7fffdf3f3180
Receiving Data:key3.0x7fffdf3f3180
Receiving Data:last
Receiving Data:key1.0x7fffe4c7fa10
Receiving Data:key2.0x7fffe4c7fa10
Receiving Data:key3.0x7fffe4c7fa10
Receiving Data:last
```

6.3. LVQ_NO_BROWSE semantics:

LVQ uses a header for a key, if the key matches it replaces the message in-place in the queue except a.) if the message with the matching key has been acquired In these two cases the message is placed into the

queue in FIFO, if another message with the same key is received it will the 'un-accessed' message with the same key will be replaced

Note, in this case browsed messaged are not invalidated, so updates can be missed.

An example

```
[localhost tests]$ ./lvqtest --mode create_lvq_no_browse
[localhost tests]$ ./lvqtest --mode write
Sending Data: key1=key1.0x7fffce5fb390
Sending Data: key2=key2.0x7fffce5fb390
Sending Data: key3=key3.0x7fffce5fb390
Sending Data: key1=key1.0x7fffce5fb390
Sending Data: last=last
[localhost tests]$ ./lvqtest --mode write
Sending Data: key1=key1.0x7fff346ae440
Sending Data: key2=key2.0x7fff346ae440
Sending Data: key3=key3.0x7fff346ae440
Sending Data: key1=key1.0x7fff346ae440
Sending Data: last=last
[localhost tests]$ ./lvqtest --mode browse
Receiving Data:key1.0x7fff346ae440
Receiving Data:key2.0x7fff346ae440
Receiving Data:key3.0x7fff346ae440
Receiving Data:last
[localhost tests]$ ./lvqtest --mode browse
Receiving Data:key1.0x7fff346ae440
Receiving Data:key2.0x7fff346ae440
Receiving Data:key3.0x7fff346ae440
Receiving Data:last
[localhost tests]$ ./lvqtest --mode write
Sending Data: key1=key1.0x7fff606583e0
Sending Data: key2=key2.0x7fff606583e0
Sending Data: key3=key3.0x7fff606583e0
Sending Data: key1=key1.0x7fff606583e0
Sending Data: last=last
[localhost tests]$ ./lvqtest --mode consume
Receiving Data:key1.0x7fff606583e0
Receiving Data:key2.0x7fff606583e0
Receiving Data:key3.0x7fff606583e0
Receiving Data:last
[localhost tests]$
```

6.4. LVQ Program Example

```
/*
 *
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
```

```
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.  See the License for the
 * specific language governing permissions and limitations
 * under the License.
 *
 */


#include <qpid/client/AsyncSession.h>
#include <qpid/client/Connection.h>
#include <qpid/client/SubscriptionManager.h>
#include <qpid/client/Session.h>
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/QueueOptions.h>

#include <iostream>

using namespace qpid::client;
using namespace qpid::framing;
using namespace qpid::sys;
using namespace qpid;
using namespace std;


enum Mode { CREATE_LVQ, CREATE_LVQ_NO_BROWSE, WRITE, BROWSE, CONSUME};
const char* modeNames[] = { "create_lvq","create_lvq_no_browse","write","browse","

// istream/ostream ops so Options can read/display Mode.
istream& operator>>(istream& in, Mode& mode) {
    string s;
    in >> s;
    int i = find(modeNames, modeNames+5, s) - modeNames;
    if (i >= 5)  throw Exception("Invalid mode: "+s);
    mode = Mode(i);
    return in;
}

ostream& operator<<(ostream& out, Mode mode) {
    return out << modeNames[mode];
}

struct  Args : public qpid::Options,
               public qpid::client::ConnectionSettings
{
```

```
    bool help;
    Mode mode;

    Args() : qpid::Options("Simple latency test optins"), help(false), mode(BROWSE
    {
        using namespace qpid;
        addOptions()
            ("help", optValue(help), "Print this usage statement")
            ("broker,b", optValue(host, "HOST"), "Broker host to connect to")
            ("port,p", optValue(port, "PORT"), "Broker port to connect to")
            ("username", optValue(username, "USER"), "user name for broker log in.
            ("password", optValue(password, "PASSWORD"), "password for broker log
            ("mechanism", optValue(mechanism, "MECH"), "SASL mechanism to use when
            ("tcp-nodelay", optValue(tcpNoDelay), "Turn on tcp-nodelay")
            ("mode", optValue(mode, "'see below'"), "Action mode."
             "\ncreate_lvq: create a new queue of type lvq.\n"
             "\ncreate_lvq_no_browse: create a new queue of type lvq with no lvq o
             "\nwrite: write a bunch of data & keys.\n"
             "\nbrowse: browse the queue.\n"
             "\nconsume: consume from the queue.\n");
    }
};

class Listener : public MessageListener
{
  private:
    Session session;
    SubscriptionManager subscriptions;
    std::string queue;
    Message request;
    QueueOptions args;
  public:
    Listener(Session& session);
    void setup(bool browse);
    void send(std::string kv);
    void received(Message& message);
    void browse();
    void consume();
};

Listener::Listener(Session& s) :
    session(s), subscriptions(s),
    queue("LVQtester")
{}

void Listener::setup(bool browse)
{
    // set queue mode
    args.setOrdering(browse?LVQ_NO_BROWSE:LVQ);

    session.queueDeclare(arg::queue=queue, arg::exclusive=false, arg::autoDelete=f

}
```

```
void Listener::browse()
{
    subscriptions.subscribe(*this, queue, SubscriptionSettings(FlowControl::unlimi
    subscriptions.run();
}

void Listener::consume()
{
    subscriptions.subscribe(*this, queue, SubscriptionSettings(FlowControl::unlimi
    subscriptions.run();
}

void Listener::send(std::string kv)
{
    request.getDeliveryProperties().setRoutingKey(queue);

    std::string key;
        args.getLVQKey(key);
    request.getHeaders().setString(key, kv);

    std::ostringstream data;
    data << kv;
    if (kv != "last") data << "." << hex << this;
    request.setData(data.str());

    cout << "Sending Data: " << kv << "=" << data.str() << std::endl;
    async(session).messageTransfer(arg::content=request);

}

void Listener::received(Message& response)
{

    cout << "Receiving Data:" << response.getData() << std::endl;
/*    if (response.getData() == "last"){
        subscriptions.cancel(queue);
    }
*/
}

int main(int argc, char** argv)
{
    Args opts;
    opts.parse(argc, argv);

    if (opts.help) {
        std::cout << opts << std::endl;
        return 0;
    }

    Connection connection;
    try {
        connection.open(opts);
        Session session = connection.newSession();
```

```
            Listener listener(session);

            switch (opts.mode)
            {
            case CONSUME:
               listener.consume();
               break;
            case BROWSE:
               listener.browse();
               break;
            case CREATE_LVQ:
               listener.setup(false);
               break;
            case CREATE_LVQ_NO_BROWSE:
               listener.setup(true);
               break;
            case WRITE:
               listener.send("key1");
               listener.send("key2");
               listener.send("key3");
               listener.send("key1");
               listener.send("last");
               break;
            }
            connection.close();
            return 0;
        } catch(const std::exception& error) {
            std::cout << error.what() << std::endl;
        }
        return 1;
    }
```

7. Queue State Replication

7.1. Asynchronous Replication of Queue State

7.1.1. Overview

There is support in qpidd for selective asynchronous replication of queue state. This is achieved by:

(a) enabling event generation for the queues in question

(b) loading a plugin on the 'source' broker to encode those events as messages on a replication queue (this plugin is called replicating_listener.so)

(c) loading a custom exchange plugin on the 'backup' broker (this plugin is called replication_exchange.so)

(d) creating an instance of the replication exchange type on the backup broker

(e) establishing a federation bridge between the replication queue on the source broker and the replication exchange on the backup broker

The bridge established between the source and backup brokers for replication (step (e) above) should have acknowledgements turned on (this may be done through the --ack N option to qpid-route). This ensures that replication events are not lost if the bridge fails.

The replication protocol will also eliminate duplicates to ensure reliably replicated state. Note though that only one bridge per replication exchange is supported. If clients try to publish to the replication exchange or if more than a the single required bridge from the replication queue on the source broker is created, replication will be corrupted. (Access control may be used to restrict access and help prevent this).

The replicating event listener plugin (step (b) above) has the following options:

```
Queue Replication Options:
  --replication-queue QUEUE                       Queue on which events for
                                                  other queues are recorded
  --replication-listener-name NAME (replicator)   name by which to register the
                                                  replicating event listener
  --create-replication-queue                      if set, the replication will
                                                  be created if it does not
                                                  exist
```

The name of the queue is required. It can either point to a durable queue whose definition has been previously recorded, or the --create-replication-queue option can be specified in which case the queue will be created a simple non-durable queue if it does not already exist.

7.1.2. Use with Clustering

The source and/or backup brokers may also be clustered brokers. In this case the federated bridge will be re-established between replicas should either of the originally connected nodes fail. There are however the following limitations at present:

- The backup site does not process membership updates after it establishes the first connection. In order for newly added members on a source cluster to be eligible as failover targets, the bridge must be recreated after those members have been added to the source cluster.
- New members added to a backup cluster will not receive information about currently established bridges. Therefore in order to allow the bridge to be re-established from these members in the event of failure of older nodes, the bridge must be recreated after the new members have joined.
- Only a single URL can be passed to create the initial link from backup site to the primary site. this means that at the time of creating the initial connection the initial node in the primary site to which the connection is made needs to be running. Once connected the backup site will receive a membership update of all the nodes in the primary site, and if the initial connection node in the primary fails, the link will be re-established on the next node that was started (time) on the primary site.

Due to the acknowledged transfer of events over the bridge (see note above) manual recreation of the bridge and automatic re-establishment of te bridge after connection failure (including failover where either or both ends are clustered brokers) will not result in event loss.

7.1.3. Operations on Backup Queues

When replicating the state of a queue to a backup broker it is important to recognise that any other operations performed directly on the backup queue may break the replication.

If the backup queue is to be an active (i.e. accessed by clients while replication is on) only enqueues should be selected for replication. In this mode, any message enqueued on the source brokers copy of the queue will also be enqueued on the backup brokers copy. However not attempt will be made to remove messages from the backup queue in response to removal of messages from the source queue.

7.1.4. Selecting Queues for Replication

Queues are selected for replication by specifying the types of events they should generate (it is from these events that the replicating plugin constructs messages which are then pulled and processed by the backup site). This is done through options passed to the initial queue-declare command that creates the queue and may be done either through qpid-config or similar tools, or by the application.

With qpid-config, the --generate-queue-events options is used:

```
--generate-queue-events N
                      If set to 1, every enqueue will generate an event that ca
                      registered listeners (e.g. for replication). If set to 2,
                      generated for enqueues and dequeues
```

From an application, the arguments field of the queue-declare AMQP command is used to convey this information. An entry should be added to the map with key 'qpid.queue_event_generation' and an integer value of 1 (to replicate only enqueue events) or 2 (to replicate both enqueue and dequeue events).

Applications written using the c++ client API may fine the qpid::client::QueueOptions class convenient. This has a enableQueueEvents() method on it that can be used to set the option (the instance of QueueOptions is then passed as the value of the arguments field in the queue-declare command. The boolean option to that method should be set to true if only enequeue events should be replicated; by default it is false meaning that both enqueues and dequeues will be replicated. E.g.

```
QueueOptions options;
options.enableQueueEvents(false);
session.queueDeclare(arg::queue="my-queue", arg::arguments=options);
```

7.1.5. Example

Lets assume we will run the primary broker on host1 and the backup on host2, have installed qpidd on both and have the replicating_listener and replication_exchange plugins in qpidd's module directory(*1).

On host1 we start the source broker and specifcy that a queue called 'replication' should be used for storing the events until consumed by the backup. We also request that this queue be created (as transient) if not already specified:

```
qpidd --replication-queue replication-queue --create-replication-queue true --
```

On host2 we start up the backup broker ensuring that the replication exchange module is loaded:

```
qpidd
```

We can then create the instance of that replication exchange that we will use to process the events:

```
qpid-config -a host2 add exchange replication replication-exchange
```

If this fails with the message "Exchange type not implemented: replication", it means the replication exchange module was not loaded. Check that the module is installed on your system and if necessary provide the full path to the library.

We then connect the replication queue on the source broker with the replication exchange on the backup broker using the qpid-route command:

```
qpid-route --ack 50 queue add host2 host1 replication-exchange replication-que
```

The example above configures the bridge to acknowledge messages in batches of 50.

Now create two queues (on both source and backup brokers), one replicating both enqueues and dequeues (queue-a) and the other replicating only dequeues (queue-b):

```
qpid-config -a host1 add queue queue-a --generate-queue-events 2
qpid-config -a host1 add queue queue-b --generate-queue-events 1

qpid-config -a host2 add queue queue-a
qpid-config -a host2 add queue queue-b
```

We are now ready to use the queues and see the replication.

Any message enqueued on queue-a will be replicated to the backup broker. When the message is acknowledged by a client connected to host1 (and thus dequeued), that message will be removed from the copy of the queue on host2. The state of queue-a on host2 will thus mirror that of the equivalent queue on host1, albeit with a small lag. (Note however that we must not have clients connected to host2 publish to- or consume from- queue-a or the state will fail to replicate correctly due to conflicts).

Any message enqueued on queue-b on host1 will also be enqueued on the equivalent queue on host2. However the acknowledgement and consequent dequeuing of messages from queue-b on host1 will have no effect on the state of queue-b on host2.

(*1) If not the paths in the above may need to be modified. E.g. if using modules built from a qpid svn checkout, the following would be added to the command line used to start qpidd on host1:

```
--load-module <path-to-qpid-dir>/src/.libs/replicating_listener.so
```

and the following for the equivalent command line on host2:

```
--load-module <path-to-qpid-dir>/src/.libs/replication_exchange.so
```

8. Starting a cluster

8.1. Running a Qpidd cluster

There are several pre-requisites to running a qpidd cluster:

8.1.1. Install and configure openais/corosync

Qpid clustering uses a multicast protocol provided by the corosync (formerly called openais) library. Install whichever is available on your OS. E.g. in fedora10: yum install corosync.

The configuration file is /etc/ais/openais.conf on openais, /etc/corosync.conf on early corosync versions and /etc/corosync/corosync.conf on recent corosync versions. You will need to edit the default file created when you installed

Here is an example, with places marked that you will change. (Below, I will describe how to change the file.)

```
# Please read the openais.conf.5 manual page

totem {
        version: 2
        secauth: off
        threads: 0
        interface {
                ringnumber: 0
                ## You must change this address ##
                bindnetaddr: 20.0.100.0
                mcastaddr: 226.94.32.36
                mcastport: 5405
        }
}

logging {
        debug: off
        timestamp: on
        to_file: yes
        logfile: /tmp/aisexec.log
}

amf {
        mode: disabled
}
```

You must sent the bindnetaddr entry in the configuration file to the network address of your network interface. This must be a real network interface, not the loopback address 127.0.0.1

You can find your network interface by running ifconfig. This will list the address and the mask, e.g.

```
inet addr:20.0.20.32  Bcast:20.0.20.255  Mask:255.255.255.0
```

The bindnetaddr is the logical AND of the inet addr and mask values, in the example above 20.0.20.0

8.1.2. Open your firewall

In the above example file, I use mcastport 5405. This implies that your firewall must allow UDP protocol over port 5405, or that you disable the firewall

8.1.3. Use the proper identity.

The qpidd process must be started with the correct identity in order to use the corosync/openais library.

For openais and early corosync versions the installation of openAIS/corosync on your system will create a new group called "ais". The user that starts the qpidd processes of the cluster must have "ais" as its effective group id. You can create a user specifically for this purpose with ais as the primary group, or a user that has ais as a secondary group can use "newgrp" to set the primary group to ais when running qpidd.

For recent corosync versions you no longer need to set your group to "ais" but you do need to create a file in /etc/corosync/uidgid.d/ to allow access for whatever user/group ID you want to use. For example create /etc/corosync/uidgid.d/qpid th the contents:

```
uidgid {
   uid: qpid
   gid: qpid
}
```

8.1.4. Starting a Cluster

To be a member of a cluster you must pass the --cluster-name argument to qpidd. This is the only required option to join a cluster, other options can be set as for a normal qpidd.

For example to start a cluster of 3 brokers on the current host Here is an example of starting a cluster of 3 members, all on the current host but with different ports and different log files:

```
qpidd -p5672 --cluster-name=MY_CLUSTER --log-output=cluster0.log -d --no-data-dir
qpidd -p5673 --cluster-name=MY_CLUSTER --log-output=cluster0.log -d --no-data-dir
qpidd -p5674 --cluster-name=MY_CLUSTER --log-output=cluster0.log -d --no-data-dir
```

In a deployed system, cluster members will normally be on different hosts but for development its useful to be able to create a cluster on a single host.

8.1.5. SELinux conflicts

Developers will often start openais/corosync as a service like this:

service openais start

But will then will start a cluster-broker without using the service script like this:

/usr/sbin/qpidd --cluster-name my_cluster ...

If SELinux is in enforcing mode this may cause qpidd to hang due because of the different SELinux contexts. There are 3 ways to resolve this:

- run both qpidd and openais/corosync as services.
- run both qpidd and openais/corosync as user processes.

- make selinux permissive:

To check what mode selinux is running:

```
# getenforce
```

To change the mode:

```
# setenforce permissive
```

Note that in a deployed system both openais/corosync and qpidd should be started as services, in which case there is no problem with SELinux running in enforcing mode.

8.1.6. Troubleshooting checklist.

If you have trouble starting your cluster, make sure that:

1. You have edited the correct openais/corosync configuration file and set bindnetaddr correctly 1. Your firewall allows UDP on the openais/corosync mcastport 2. Your effective group is "ais" (openais/old corosync) or you have created an appropriate ID file (new corosync) 3. Your firewall allows TCP on the ports used by qpidd. 4. If you're starting openais as a service but running qpidd directly, ensure selinux is in permissive mode

9. ACL

9.1. v2 ACL file format for brokers

This new ACL implementation has been designed for implementation and interoperability on all Qpid brokers. It is currently supported in the following brokers:

Table 5.2. ACL Support in Qpid Broker Versions

Broker	Version
C++	M4 onward
Java	M5 anticipated

Contents

- • Section 9.3.1, “ Writing Good/Fast ACL ”
  - Section 9.3.2, “ Getting ACL to Log ”
  - Section 9.3.3, “ User Id / domains running with C++ broker ”

9.1.1. Specification

Notes on file formats

- A line starting with the character '#' will be considered a comment, and are ignored.
- Since the '#' char (and others that are commonly used for comments) are commonly found in routing keys and other AMQP literals, it is simpler (for now) to hold off on allowing trailing comments (ie comments in which everything following a '#' is considered a comment). This could be reviewed later once the rest of the format is finalized.
- Empty lines ("") and lines that contain only whitespace (any combination of ' ', '\f', '\n', '\r', '\t', '\v') are ignored.
- All tokens are case sensitive. "name1" != "Name1" and "create" != "CREATE".
- Group lists may be extended to the following line by terminating the line with the '\' character. However, this may only occur after the group name or any of the names following the group name. Empty extension lines (ie just a '\' character) are not permitted.

```
# Examples of extending group lists using a trailing '\' character

group group1 name1 name2 \
             name3 name4 \
             name5

group group2 \
             group1 \
             name6

# The following are illegal:

# '\' must be after group name
group \
      group3 name7 name8

# No empty extension lines
group group4 name9 \
                   \
             name10
```

- Additional whitespace (ie more than one whitespace char) between and after tokens is ignored. However group and acl definitions must start with "group" or "acl" respectively and with no preceding whitespace.
- All acl rules are limited to a single line.
- Rules are interpreted from the top of the file down until the name match is obtained; at which point processing stops.

- The keyword "all" is reserved, and matches all individuals, groups and actions. It may be used in place of a group or individual name and/or an action - eg "acl allow all all", "acl deny all all" or "acl deny user1 all".
- The last line of the file (whether present or not) will be assumed to be "acl deny all all". If present in the file, any lines below this one are ignored.
- Names and group names may contain only a-z, A-Z, 0-9, '-','_'.
- Rules must be preceded by any group definitions they may use; any name not previously defined as a group will be assumed to be that of an individual.
- ACL rules must have the following tokens in order on a single line:
  - The string literal "acl";
  - The permission;
  - The name of a single group or individual or the keyword "all";
  - The name of an action or the keyword "all";
  - Optionally, a single object name or the keyword "all";
  - If the object is present, then optionally one or more property name-value pair(s) (in the form property=value).

```
user = username[@domain[/realm]]
user-list = user1 user2 user3 ...
group-name-list = group1 group2 group3 ...

group <group-name> = [user-list] [group-name-list]


permission = [allow|allow-log|deny|deny-log]
action = [consume|publish|create|access|bind|unbind|delete|purge|update]
object = [virtualhost|queue|exchange|broker|link|route|method]
property = [name|durable|owner|routingkey|passive|autodelete|exclusive|type|altern

acl permission {<group-name>|<user-name>|"all"} {action|"all"} [object|"all"] [pro
```

9.1.2. Validation

The new ACL file format needs to perform validation on the acl rules. The validation should be performed depending on the set value:

strict-acl-validation=none The default setting should be 'warn'

On validation of this acl the following checks would be expected:

```
acl allow client publish routingkey=exampleQueue exchange=amq.direct
```

1. The If the user 'client' cannot be found, if the authentication mechanism cannot be queried then a 'user' value should be added to the file.

2. There is an exchange called 'amq.direct'

3. There is a queue bound to 'exampleQueue' on 'amq.direct'

Each of these checks that fail will result in a log statement being generated.

In the case of a fatal logging the full file will be validated before the broker shuts down.

9.1.3. Example file:

```
# Some groups
group admin ted@QPID martin@QPID
group user-consume martin@QPID ted@QPID
group group2 kim@QPID user-consume rob@QPID
group publisher group2 \
                tom@QPID andrew@QPID debbie@QPID

# Some rules
acl allow carlt@QPID create exchange name=carl.*
acl deny rob@QPID create queue
acl allow guest@QPID bind exchange name=amq.topic routingkey=stocks.ibm.#  owner=s
acl allow user-consume create queue name=tmp.*

acl allow publisher publish all durable=false
acl allow publisher create queue name=RequestQueue
acl allow consumer consume queue durable=true
acl allow fred@QPID create all
acl allow bob@QPID all queue
acl allow admin all
acl deny kim@QPID all
acl allow all consume queue owner=self
acl allow all bind exchange owner=self

# Last (default) rule
acl deny all all
```

9.2. Design Documentation

9.2.1. Mapping of ACL traps to action and type

The C++ broker maps the ACL traps in the follow way for AMQP 0-10: The Java broker currently only performs ACLs on the AMQP connection not on management functions:

Table 5.3. Mapping ACL Traps

Object	Action	Properties	Trap C++	Trap Java
Exchange	Create	name type alternate passive durable	ExchangeHandlerImpl::create	ExchangeDeclareHandler
Exchange	Delete	name	ExchangeHandlerImpl::delete	ExchangeDeleteHandler
Exchange	Access	name	ExchangeHandlerImpl::query	

Object	Action	Properties	Trap C++	Trap Java
Exchange	Bind	name routingkey queuename owner	ExchangeHandlerImpl::bind	QueueBindHandler
Exchange	Unbind	name routingkey	ExchangeHandlerImpl::unbind	ExchangeUnbindHandler
Exchange	Access	name queuename routingkey	ExchangeHandlerImpl::bound	
Exchange	Publish	name routingKey	SemanticState::route	BasicPublishMethodHandler
Queue	Access	name	QueueHandlerImpl::query	
Queue	Create	name alternate passive durable exclusive autodelete	QueueHandlerImpl::declare	QueueDeclareHandler
Queue	Purge	name	QueueHandlerImpl::purge	QueuePurgeHandler
Queue	Purge	name	Management::Queue::purge	
Queue	Delete	name	QueueHandlerImpl::delete	QueueDeleteHandler
Queue	Consume	name (possibly add in future?)	MessageHandlerImpl::consume	BasicConsumeMethodHandler BasicGetMethodHandler
<Object>	Update		ManagementProperty::set	
<Object>	Access		ManagementProperty::read	
Link	Create		Management::connect	
Route	Create		Management:: -createFederationRoute-	
Route	Delete		Management:: -deleteFederationRoute-	
Virtualhost	Access	name	TBD	ConnectionOpenMethodHandler

Management actions that are not explicitly given a name property it will default the name property to management method name, if the action is 'W' Action will be 'Update', if 'R' Action will be 'Access'.

for example, if the mgnt method 'joinCluster' was not mapped in schema it will be mapped in ACL file as follows

Table 5.4. Mapping Management Actions to ACL

Object	Action	Property
Broker	Update	name=joinCluster

9.3. v2 ACL User Guide

9.3.1. Writing Good/Fast ACL

The file gets read top down and rule get passed based on the first match. In the following example the first rule is a dead rule. I.e. the second rule is wider than the first rule. DON'T do this, it will force extra analysis, worst case if the parser does not kill the dead rule you might get a false deny.

```
allow peter@QPID create queue name=tmp <-- dead rule!!
```

```
allow peter@QPID create queue
deny all all
```

By default files end with

```
deny all all
```

the mode of the ACL engine can be swapped to be allow based by putting the following at the end of the file

```
allow all all
```

Note that 'allow' based file will be a LOT faster for message transfer. This is because the AMQP specification does not allow for creating subscribes on publish, so the ACL is executed on every message transfer. Also, ACL's rules using less properties on publish will in general be faster.

9.3.2. Getting ACL to Log

In order to get log messages from ACL actions use allow-log and deny-log for example

```
allow-log john@QPID all all
deny-log guest@QPID all all
```

9.3.3. User Id / domains running with C++ broker

The user-id used for ACL is taken from the connection user-id. Thus in order to use ACL the broker authentication has to be setup. i.e. (if --auth no is used in combination with ACL the broker will deny everything)

The user id in the ACL file is of the form <user-id>@<domain> The Domain is configured via the SASL configuration for the broker, and the domain/realm for qpidd is set using --realm and default to 'QPID'.

To load the ACL module use, load the acl module cmd line or via the config file

```
./src/qpidd --load-module src/.libs/acl.so
```

The ACL plugin provides the following option '--acl-file'. If do ACL file is supplied the broker will not enforce ACL. If an ACL file name is supplied, and the file does not exist or is invalid the broker will not start.

```
ACL Options:
  --acl-file FILE       The policy file to load from, loaded from data dir
```

Chapter 6. Managing the AMQP Messaging Broker

1. Managing the C++ Broker

There are quite a few ways to interact with the C++ broker. The command line tools include:

- qpid-route - used to configure federation (a set of federated brokers)
- qpid-config - used to configure queues, exchanges, bindings and list them etc
- qpid-tool - used to view management information/statistics and call any management actions on the broker
- qpid-printevents - used to receive and print QMF events

1.1. Using qpid-config

This utility can be used to create queues exchanges and bindings, both durable and transient. Always check for latest options by running --help command.

```
$ qpid-config --help
Usage:  qpid-config [OPTIONS]
        qpid-config [OPTIONS] exchanges [filter-string]
        qpid-config [OPTIONS] queues    [filter-string]
        qpid-config [OPTIONS] add exchange <type> <name> [AddExchangeOptions]
        qpid-config [OPTIONS] del exchange <name>
        qpid-config [OPTIONS] add queue <name> [AddQueueOptions]
        qpid-config [OPTIONS] del queue <name>
        qpid-config [OPTIONS] bind   <exchange-name> <queue-name> [binding-key]
        qpid-config [OPTIONS] unbind <exchange-name> <queue-name> [binding-key]

Options:
    -b [ --bindings ]                         Show bindings in queue or exchange l
    -a [ --broker-addr ] Address (localhost)  Address of qpidd broker
         broker-addr is in the form:   [username/password@] hostname | ip-address
         ex:  localhost, 10.1.1.7:10000, broker-host:10000, guest/guest@localhost

Add Queue Options:
    --durable            Queue is durable
    --cluster-durable    Queue becomes durable if there is only one functioning cl
    --file-count N (8)   Number of files in queue's persistence journal
    --file-size  N (24)  File size in pages (64Kib/page)
    --max-queue-size N   Maximum in-memory queue size as bytes
    --max-queue-count N  Maximum in-memory queue size as a number of messages
    --limit-policy [none | reject | flow-to-disk | ring | ring-strict]
                         Action taken when queue limit is reached:
                             none (default) - Use broker's default policy
                             reject         - Reject enqueued messages
                             flow-to-disk   - Page messages to disk
                             ring           - Replace oldest unacquired message wi
```

```
                              ring-strict    - Replace oldest message, reject if ol
    --order [fifo | lvq | lvq-no-browse]
                          Set queue ordering policy:
                              fifo (default) - First in, first out
                              lvq            - Last Value Queue ordering, allows qu
                              lvq-no-browse  - Last Value Queue ordering, browsing
    --generate-queue-events N
                          If set to 1, every enqueue will generate an event that ca
                          registered listeners (e.g. for replication). If set to 2,
                          generated for enqueues and dequeues

Add Exchange Options:
    --durable    Exchange is durable
    --sequence   Exchange will insert a 'qpid.msg_sequence' field in the message h
                 with a value that increments for each message forwarded.
    --ive        Exchange will behave as an 'initial-value-exchange', keeping a re
                 to the last message forwarded and enqueuing that message to newly
                 queues.
```

Get the summary page

```
$ qpid-config
Total Exchanges: 6
          topic: 2
        headers: 1
         fanout: 1
         direct: 2
   Total Queues: 7
        durable: 0
    non-durable: 7
```

List the queues

```
$ qpid-config queues
Queue Name                                        Attributes
================================================================
pub_start
pub_done
sub_ready
sub_done
perftest0                                         --durable
reply-dhcp-100-18-254.bos.redhat.com.20713  auto-del excl
topic-dhcp-100-18-254.bos.redhat.com.20713  auto-del excl
```

List the exchanges with bindings

```
$ ./qpid-config -b exchanges
Exchange '' (direct)
    bind pub_start => pub_start
    bind pub_done => pub_done
    bind sub_ready => sub_ready
    bind sub_done => sub_done
```

```
    bind perftest0 => perftest0
    bind mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => mgmt-3206ff16-fb29-4a30-82ea
    bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-4a30-82ea
Exchange 'amq.direct' (direct)
    bind repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15 => repl-3206ff16-fb29-4a30-82ea
    bind repl-df06c7a6-4ce7-426a-9f66-da91a2a6a837 => repl-df06c7a6-4ce7-426a-9f66
    bind repl-c55915c2-2fda-43ee-9410-b1c1cbb3e4ae => repl-c55915c2-2fda-43ee-9410
Exchange 'amq.topic' (topic)
Exchange 'amq.fanout' (fanout)
Exchange 'amq.match' (headers)
Exchange 'qpid.management' (topic)
    bind mgmt.# => mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15
```

1.2. Using qpid-route

This utility is to create federated networks of brokers, This allows you for forward messages between brokers in a network. Messages can be routed statically (using "qpid-route route add") where the bindings that control message forwarding are supplied in the route. Message routing can also be dynamic (using "qpid-route dynamic add") where the messages are automatically forwarded to clients based on their bindings to the local broker.

```
$ qpid-route
Usage:  qpid-route [OPTIONS] dynamic add <dest-broker> <src-broker> <exchange> [ta
        qpid-route [OPTIONS] dynamic del <dest-broker> <src-broker> <exchange>

        qpid-route [OPTIONS] route add   <dest-broker> <src-broker> <exchange> <ro
        qpid-route [OPTIONS] route del   <dest-broker> <src-broker> <exchange> <ro
        qpid-route [OPTIONS] queue add   <dest-broker> <src-broker> <exchange> <qu
        qpid-route [OPTIONS] queue del   <dest-broker> <src-broker> <exchange> <qu
        qpid-route [OPTIONS] route list  [<dest-broker>]
        qpid-route [OPTIONS] route flush [<dest-broker>]
        qpid-route [OPTIONS] route map   [<broker>]

        qpid-route [OPTIONS] link add  <dest-broker> <src-broker>
        qpid-route [OPTIONS] link del  <dest-broker> <src-broker>
        qpid-route [OPTIONS] link list [<dest-broker>]

Options:
    -v [ --verbose ]         Verbose output
    -q [ --quiet ]           Quiet output, don't print duplicate warnings
    -d [ --durable ]         Added configuration shall be durable
    -e [ --del-empty-link ]  Delete link after deleting last route on the link
    -s [ --src-local ]       Make connection to source broker (push route)
    -t <transport> [ --transport <transport>]
                             Specify transport to use for links, defaults to tcp

  dest-broker and src-broker are in the form:  [username/password@] hostname | ip-
  ex:  localhost, 10.1.1.7:10000, broker-host:10000, guest/guest@localhost
```

A few examples:

```
qpid-route dynamic add host1 host2 fed.topic
```

```
qpid-route dynamic add host2 host1 fed.topic

qpid-route -v route add host1 host2 hub1.topic hub2.topic.stock.buy
qpid-route -v route add host1 host2 hub1.topic hub2.topic.stock.sell
qpid-route -v route add host1 host2 hub1.topic 'hub2.topic.stock.#'
qpid-route -v route add host1 host2 hub1.topic 'hub2.#'
qpid-route -v route add host1 host2 hub1.topic 'hub2.topic.#'
qpid-route -v route add host1 host2 hub1.topic 'hub2.global.#'
```

The link map feature can be used to display the entire federated network configuration by supplying a single broker as an entry point:

```
$ qpid-route route map localhost:10001

Finding Linked Brokers:
    localhost:10001... Ok
    localhost:10002... Ok
    localhost:10003... Ok
    localhost:10004... Ok
    localhost:10005... Ok
    localhost:10006... Ok
    localhost:10007... Ok
    localhost:10008... Ok

Dynamic Routes:

  Exchange fed.topic:
    localhost:10002 <=> localhost:10001
    localhost:10003 <=> localhost:10002
    localhost:10004 <=> localhost:10002
    localhost:10005 <=> localhost:10002
    localhost:10006 <=> localhost:10005
    localhost:10007 <=> localhost:10006
    localhost:10008 <=> localhost:10006

  Exchange fed.direct:
    localhost:10002  => localhost:10001
    localhost:10004  => localhost:10003
    localhost:10003  => localhost:10002
    localhost:10001  => localhost:10004

Static Routes:

  localhost:10003(ex=amq.direct) <= localhost:10005(ex=amq.direct) key=rkey
  localhost:10003(ex=amq.direct) <= localhost:10005(ex=amq.direct) key=rkey2
```

1.3. Using qpid-tool

This utility provided a telnet style interface to be able to view, list all stats and action all the methods. Simple capture below. Best to just play with it and mail the list if you have questions or want features added.

```
qpid:
```

```
qpid: help
Management Tool for QPID
Commands:
    list                              - Print summary of existing objects by class
    list <className>                  - Print list of objects of the specified class
    list <className> all              - Print contents of all objects of specified c
    list <className> active           - Print contents of all non-deleted objects of
    list <list-of-IDs>                - Print contents of one or more objects (infer
    list <className> <list-of-IDs>    - Print contents of one or more objects
        list is space-separated, ranges may be specified (i.e. 1004-1010)
    call <ID> <methodName> <args> - Invoke a method on an object
    schema                            - Print summary of object classes seen on the
    schema <className>                - Print details of an object class
    set time-format short             - Select short timestamp format (default)
    set time-format long              - Select long timestamp format
    quit or ^D                        - Exit the program
qpid: list
Management Object Types:
    ObjectType     Active  Deleted
    ================================
    qpid.binding   21      0
    qpid.broker    1       0
    qpid.client    1       0
    qpid.exchange  6       0
    qpid.queue     13      0
    qpid.session   4       0
    qpid.system    1       0
    qpid.vhost     1       0
qpid: list qpid.system
Objects of type qpid.system
    ID    Created   Destroyed  Index
    ================================
    1000  21:00:02  -          host
qpid: list 1000
Object of type qpid.system: (last sample time: 21:26:02)
    Type    Element   1000
    ======================================================
    config  sysId     host
    config  osName    Linux
    config  nodeName  localhost.localdomain
    config  release   2.6.24.4-64.fc8
    config  version   #1 SMP Sat Mar 29 09:15:49 EDT 2008
    config  machine   x86_64
qpid: schema queue
Schema for class 'qpid.queue':
    Element                Type          Unit         Access      Notes   Descript
    ============================================================================
    vhostRef               reference                  ReadCreate  index
    name                   short-string               ReadCreate  index
    durable                boolean                    ReadCreate
    autoDelete             boolean                    ReadCreate
    exclusive              boolean                    ReadCreate
    arguments              field-table                ReadOnly            Argument
    storeRef               reference                  ReadOnly            Referenc
```

```
    msgTotalEnqueues       uint64       message                        Total me
    msgTotalDequeues       uint64       message                        Total me
    msgTxnEnqueues         uint64       message                        Transact
    msgTxnDequeues         uint64       message                        Transact
    msgPersistEnqueues     uint64       message                        Persiste
    msgPersistDequeues     uint64       message                        Persiste
    msgDepth               uint32       message                        Current
    msgDepthHigh           uint32       message                        Current
    msgDepthLow            uint32       message                        Current
    byteTotalEnqueues      uint64       octet                          Total me
    byteTotalDequeues      uint64       octet                          Total me
    byteTxnEnqueues        uint64       octet                          Transact
    byteTxnDequeues        uint64       octet                          Transact
    bytePersistEnqueues    uint64       octet                          Persiste
    bytePersistDequeues    uint64       octet                          Persiste
    byteDepth              uint32       octet                          Current
    byteDepthHigh          uint32       octet                          Current
    byteDepthLow           uint32       octet                          Current
    enqueueTxnStarts       uint64       transaction                    Total en
    enqueueTxnCommits      uint64       transaction                    Total en
    enqueueTxnRejects      uint64       transaction                    Total en
    enqueueTxnCount        uint32       transaction                    Current
    enqueueTxnCountHigh    uint32       transaction                    Current
    enqueueTxnCountLow     uint32       transaction                    Current
    dequeueTxnStarts       uint64       transaction                    Total de
    dequeueTxnCommits      uint64       transaction                    Total de
    dequeueTxnRejects      uint64       transaction                    Total de
    dequeueTxnCount        uint32       transaction                    Current
    dequeueTxnCountHigh    uint32       transaction                    Current
    dequeueTxnCountLow     uint32       transaction                    Current
    consumers              uint32       consumer                       Current
    consumersHigh          uint32       consumer                       Current
    consumersLow           uint32       consumer                       Current
    bindings               uint32       binding                        Current
    bindingsHigh           uint32       binding                        Current
    bindingsLow            uint32       binding                        Current
    unackedMessages        uint32       message                        Messages
    unackedMessagesHigh    uint32       message                        Messages
    unackedMessagesLow     uint32       message                        Messages
    messageLatencySamples  delta-time   nanosecond                     Broker l
    messageLatencyMin      delta-time   nanosecond                     Broker l
    messageLatencyMax      delta-time   nanosecond                     Broker l
    messageLatencyAverage  delta-time   nanosecond                     Broker l
Method 'purge' Discard all messages on queue
qpid: list queue
Objects of type qpid.queue
    ID    Created   Destroyed  Index
    ================================================================================
    1012  21:08:13  -          1002.pub_start
    1014  21:08:13  -          1002.pub_done
    1016  21:08:13  -          1002.sub_ready
    1018  21:08:13  -          1002.sub_done
    1020  21:08:13  -          1002.perftest0
    1038  21:09:08  -          1002.mgmt-3206ff16-fb29-4a30-82ea-e76f50dd7d15
```

```
    1040  21:09:08  -          1002.repl-3206ff16-fb29-4a30-82ea-e76f50dd7d15
    1046  21:09:32  -          1002.mgmt-df06c7a6-4ce7-426a-9f66-da91a2a6a837
    1048  21:09:32  -          1002.repl-df06c7a6-4ce7-426a-9f66-da91a2a6a837
    1054  21:10:01  -          1002.mgmt-c55915c2-2fda-43ee-9410-b1c1cbb3e4ae
    1056  21:10:01  -          1002.repl-c55915c2-2fda-43ee-9410-b1c1cbb3e4ae
    1063  21:26:00  -          1002.mgmt-8d621997-6356-48c3-acab-76a37081d0f3
    1065  21:26:00  -          1002.repl-8d621997-6356-48c3-acab-76a37081d0f3
qpid: list 1020
Object of type qpid.queue: (last sample time: 21:26:02)
    Type    Element                1020
    ================================================================================
    config  vhostRef               1002
    config  name                   perftest0
    config  durable                False
    config  autoDelete             False
    config  exclusive              False
    config  arguments              {'qpid.max_size': 0, 'qpid.max_count': 0}
    config  storeRef               NULL
    inst    msgTotalEnqueues       500000 messages
    inst    msgTotalDequeues       500000
    inst    msgTxnEnqueues         0
    inst    msgTxnDequeues         0
    inst    msgPersistEnqueues     0
    inst    msgPersistDequeues     0
    inst    msgDepth               0
    inst    msgDepthHigh           0
    inst    msgDepthLow            0
    inst    byteTotalEnqueues      512000000 octets
    inst    byteTotalDequeues      512000000
    inst    byteTxnEnqueues        0
    inst    byteTxnDequeues        0
    inst    bytePersistEnqueues    0
    inst    bytePersistDequeues    0
    inst    byteDepth              0
    inst    byteDepthHigh          0
    inst    byteDepthLow           0
    inst    enqueueTxnStarts       0 transactions
    inst    enqueueTxnCommits      0
    inst    enqueueTxnRejects      0
    inst    enqueueTxnCount        0
    inst    enqueueTxnCountHigh    0
    inst    enqueueTxnCountLow     0
    inst    dequeueTxnStarts       0
    inst    dequeueTxnCommits      0
    inst    dequeueTxnRejects      0
    inst    dequeueTxnCount        0
    inst    dequeueTxnCountHigh    0
    inst    dequeueTxnCountLow     0
    inst    consumers              0 consumers
    inst    consumersHigh          0
    inst    consumersLow           0
    inst    bindings               1 binding
    inst    bindingsHigh           1
    inst    bindingsLow            1
```

```
    inst    unackedMessages        0 messages
    inst    unackedMessagesHigh    0
    inst    unackedMessagesLow     0
    inst    messageLatencySamples  0
    inst    messageLatencyMin      0
    inst    messageLatencyMax      0
    inst    messageLatencyAverage  0
qpid:
```

1.4. Using qpid-printevents

This utility connects to one or more brokers and collects events, printing out a line per event.

```
$ qpid-printevents --help
Usage: qpid-printevents [options] [broker-addr]...

Collect and print events from one or more Qpid message brokers.  If no broker-
addr is supplied, qpid-printevents will connect to 'localhost:5672'. broker-
addr is of the form:  [username/password@] hostname | ip-address [:<port>] ex:
localhost, 10.1.1.7:10000, broker-host:10000, guest/guest@localhost

Options:
  -h, --help  show this help message and exit
```

You get the idea... have fun!

2. QMan - Qpid Management bridge

2.1. QMan : Qpid Management Bridge

QMan is a management bridge for Qpid. It allows external clients to manage and monitor one or more Qpid brokers.

Please note: All WS-DM related concerns have to be considered part of M5 release.

QMan exposes the broker management interfaces using Java Management Extensions (JMX) and / or OASIS Web Services Distributed Management (WSDM). While the first one is supposed to be used by java based clients only the latter is an interoperable protocol that enables management clients to access and receive notifications of management-enabled resources using Web Services.

QMan can be easily integrated in your preexisting system in different ways :

- As a standalone application : in this case it runs as a server. More specifically it enables communication via RMI (for JMX) or via HTTP (for WS-DM); Note that when the WS-DM adapter is used the JMX interface is not exposed;
- As a deployable unit : it is also available as a standard Java web application (war); This is useful when there's a preexisting Application Server in your environment and you don't want start another additional server in order to run QMan.

2.1.1. User Documentation

With "User Documentation" we mean all information that you need to know in order to use QMan from a user perspective. Those information include :

Table 6.1.

Section	Description
???	How to install & start QMan.
???	QMan (WS-DM version only) Administration Console.
???	Describes each JMX interface exposed by QMan.
???	Describes each WS-DM interface exposed by QMan.
???	Informational / Debug / Error / Warning messages catalogue.

2.1.2. Technical Documentation

If you are interested in technical details about QMan and related technologies this is a good starting point. In general this section provides information about QMan design, interfaces, patterns and so on...

Table 6.2.

Section	Description
???	A short introduction about QMan deployment context.
???	Describes QMan components, their interactions and responsibilities.

3. Qpid Management Framework

- Section 3.1, “ What Is QMF ”
- Section 3.2, “ Getting Started with QMF ”
- Section 3.3, “ QMF Concepts ”
- • Section 3.3.1, “ Console, Agent, and Broker ”
  - Section 3.3.2, “ Schema ”
  - Section 3.3.3, “ Class Keys and Class Versioning ”
- Section 3.4, “ The QMF Protocol ”
- Section 3.5, “ How to Write a QMF Console ”
- Section 3.6, “ How to Write a QMF Agent ”

Please visit the ??? for information about the future of QMF.

3.1. What Is QMF

QMF (Qpid Management Framework) is a general-purpose management bus built on Qpid Messaging. It takes advantage of the scalability, security, and rich capabilities of Qpid to provide flexible and easy-to-use manageability to a large set of applications.

3.2. Getting Started with QMF

QMF is used through two primary APIs. The *console* API is used for console applications that wish to access and manipulate manageable components through QMF. The *agent* API is used for application that wish to be managed through QMF.

The fastest way to get started with QMF is to work through the "How To" tutorials for consoles and agents. For a deeper understanding of what is happening in the tutorials, it is recommended that you look at the *Qmf Concepts* section.

3.3. QMF Concepts

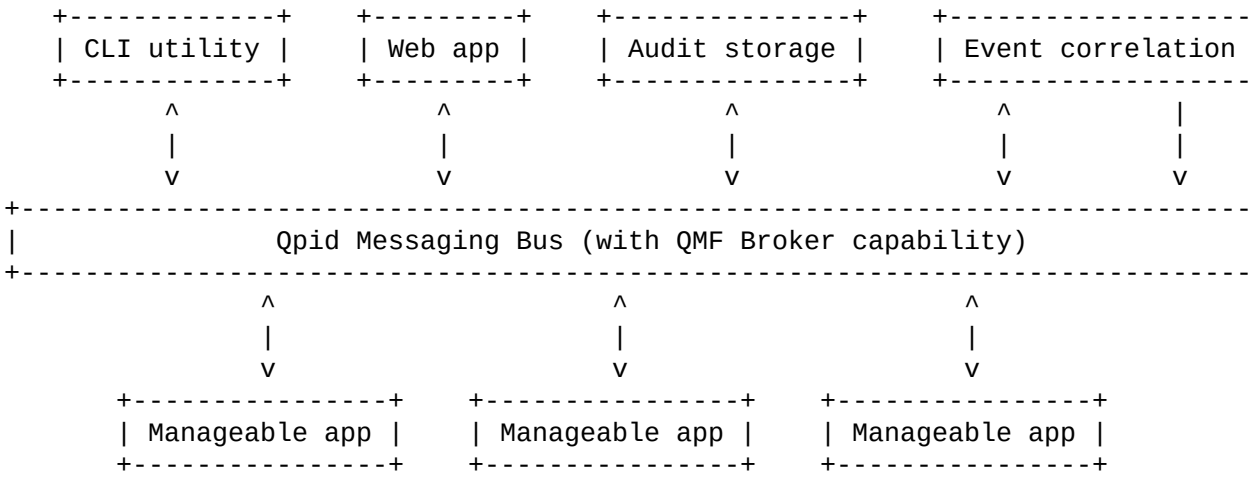
This section introduces important concepts underlying QMF.

3.3.1. Console, Agent, and Broker

The major architectural components of QMF are the Console, the Agent, and the Broker. Console components are the "managing" components of QMF and agent components are the "managed" parts. The broker is a central (possibly distributed, clustered and fault-tolerant) component that manages name spaces and caches schema information.

A console application may be a command-line utility, a three-tiered web-based GUI, a collection and storage device, a specialized application that monitors and reacts to events and conditions, or anything else somebody wishes to develop that uses QMF management data.

An agent application is any application that has been enhanced to allow itself to be managed via QMF.



In the above diagram, the *Manageable apps* are agents, the *CLI utility*, *Web app*, and *Audit storage* are consoles, and *Event correlation* is both a console and an agent because it can create events based on the aggregation of what it sees.

3.3.2. Schema

A *schema* describes the structure of management data. Each *agent* provides a schema that describes its management model including the object classes, methods, events, etc. that it provides. In the current QMF

distribution, the agent's schema is codified in an XML document. In the near future, there will also be ways to programatically create QMF schemata.

3.3.2.1. Package

Each agent that exports a schema identifies itself using a *package* name. The package provides a unique namespace for the classes in the agent's schema that prevent collisions with identically named classes in other agents' schemata.

Package names are in "reverse domain name" form with levels of hierarchy separated by periods. For example, the Qpid messaging broker uses package "org.apache.qpid.broker" and the Access Control List plugin for the broker uses package "org.apache.qpid.acl". In general, the package name should be the reverse of the internet domain name assigned to the organization that owns the agent software followed by identifiers to uniquely identify the agent.

The XML document for a package's schema uses an enclosing <schema> tag. For example:

```
<schema package="org.apache.qpid.broker">

</schema>
```

3.3.2.2. Object Classes

Object classes define types for manageable objects. The agent may create and destroy objects which are instances of object classes in the schema. An object class is defined in the XML document using the <class> tag. An object class is composed of properties, statistics, and methods.

```
<class name="Exchange">
  <property name="vhostRef"   type="objId" references="Vhost" access="RC" index=
  <property name="name"       type="sstr"  access="RC" index="y"/>
  <property name="type"       type="sstr"  access="RO"/>
  <property name="durable"    type="bool"  access="RC"/>
  <property name="arguments"  type="map"   access="RO" desc="Arguments supplied

  <statistic name="producerCount" type="hilo32"  desc="Current producers on exch
  <statistic name="bindingCount"  type="hilo32"  desc="Current bindings"/>
  <statistic name="msgReceives"   type="count64" desc="Total messages received"/
  <statistic name="msgDrops"      type="count64" desc="Total messages dropped (n
  <statistic name="msgRoutes"     type="count64" desc="Total routed messages"/>
  <statistic name="byteReceives"  type="count64" desc="Total bytes received"/>
  <statistic name="byteDrops"     type="count64" desc="Total bytes dropped (no m
  <statistic name="byteRoutes"    type="count64" desc="Total routed bytes"/>
</class>
```

3.3.2.3. Properties and Statistics

<property> and <statistic> tags must be placed within <schema> and </schema> tags.

Properties, statistics, and methods are the building blocks of an object class. Properties and statistics are both object attributes, though they are treated differently. If an object attribute is defining, seldom or never changes, or is large in size, it should be defined as a *property*. If an attribute is rapidly changing or is used to instrument the object (counters, etc.), it should be defined as a *statistic*.

The XML syntax for <property> and <statistic> have the following XML-attributes:

Table 6.3. XML Attributes for QMF Properties and Statistics

Attribute	<property>	<statistic>	Meaning
name	Y	Y	The name of the attribute
type	Y	Y	The data type of the attribute
unit	Y	Y	Optional unit name - use the singular (i.e. MByte)
desc	Y	Y	Description to annotate the attribute
references	Y		If the type is "objId", names the referenced class
access	Y		Access rights (RC, RW, RO)
index	Y		"y" if this property is used to uniquely identify the object. There may be more than one index property in a class
parentRef	Y		"y" if this property references an object in which this object is in a child-parent relationship.
optional	Y		"y" if this property is optional (i.e. may be NULL/not-present)
min	Y		Minimum value of a numeric attribute
max	Y		Maximum value of a numeric attribute
maxLen	Y		Maximum length of a string attribute

3.3.2.4. Methods

<method> tags must be placed within <schema> and </schema> tags.

A *method* is an invokable function to be performed on instances of the object class (i.e. a Remote Procedure Call). A <method> tag has a name, an optional description, and encloses zero or more arguments. Method arguments are defined by the <arg> tag and have a name, a type, a direction, and an optional description. The argument direction can be "I", "O", or "IO" indicating input, output, and input/output respectively. An example:

```
<method name="echo" desc="Request a response to test the path to the management
  <arg name="sequence" dir="IO" type="uint32"/>
  <arg name="body"     dir="IO" type="lstr"/>
</method>
```

3.3.2.5. Event Classes

3.3.2.6. Data Types

Object attributes, method arguments, and event arguments have data types. The data types are based on the rich data typing system provided by the AMQP messaging protocol. The following table describes the data types available for QMF:

Table 6.4. QMF Datatypes

QMF Type	Description
REF	QMF Object ID - Used to reference another QMF object.
U8	8-bit unsigned integer
U16	16-bit unsigned integer
U32	32-bit unsigned integer
U64	64-bit unsigned integer
S8	8-bit signed integer
S16	16-bit signed integer
S32	32-bit signed integer
S64	64-bit signed integer
BOOL	Boolean - True or False
SSTR	Short String - String of up to 255 bytes
LSTR	Long String - String of up to 65535 bytes
ABSTIME	Absolute time since the epoch in nanoseconds (64-bits)
DELTATIME	Delta time in nanoseconds (64-bits)
FLOAT	Single precision floating point number
DOUBLE	Double precision floating point number
UUID	UUID - 128 bits
FTABLE	Field-table - std::map in C++, dictionary in Python

In the XML schema definition, types go by different names and there are a number of special cases. This is because the XML schema is used in code-generation for the agent API. It provides options that control what kind of accessors are generated for attributes of different types. The following table enumerates the types available in the XML format, which QMF types they map to, and other special handling that occurs.

Table 6.5. XML Schema Mapping for QMF Types

XML Type	QMF Type	Accessor Style	Special Characteristics
objId	REF	Direct (get, set)	
uint8,16,32,64	U8,16,32,64	Direct (get, set)	
int8,16,32,64	S8,16,32,64	Direct (get, set)	
bool	BOOL	Direct (get, set)	
sstr	SSTR	Direct (get, set)	

lstr	LSTR	Direct (get, set)	
absTime	ABSTIME	Direct (get, set)	
deltaTime	DELTATIME	Direct (get, set)	
float	FLOAT	Direct (get, set)	
double	DOUBLE	Direct (get, set)	
uuid	UUID	Direct (get, set)	
map	FTABLE	Direct (get, set)	
hilo8,16,32,64	U8,16,32,64	Counter (inc, dec)	Generates value, valueMin, valueMax
count8,16,32,64	U8,16,32,64	Counter (inc, dec)	
mma32,64	U32,64	Direct	Generates valueMin, valueMax, valueAverage, valueSamples
mmaTime	DELTATIME	Direct	Generates valueMin, valueMax, valueAverage, valueSamples

Important

When writing a schema using the XML format, types used in <property> or <arg> must be types that have *Direct* accessor style. Any type may be used in <statistic> tags.

3.3.3. Class Keys and Class Versioning

3.4. The QMF Protocol

The QMF protocol defines the message formats and communication patterns used by the different QMF components to communicate with one another.

A description of the current version of the QMF protocol can be found at ???.

A proposal for an updated protocol based on map-messages is in progress and can be found at ???.

3.5. How to Write a QMF Console

Please see the ??? for information about using the console API with Python.

3.6. How to Write a QMF Agent

4. Management Design notes

4.1. Status of This Document

This document does not track any current development activity. It is the specification of the management framework implemented in the M3 release of the C++ broker and will be left here for user and developer reference.

Development continues on the Qpid Management Framework (QMF) for M4. If you are using M3, this is the document you need. If you are using the SVN trunk, please refer to ??? for up-to-date information.

4.2. Introduction

This document describes the management features that are used in the QPID C++ broker as of the M3 milestone. These features do not appear in earlier milestones nor are they implemented in the Java broker.

This specification is *not* a standard and is not endorsed by the AMQP working group. When such a standard is adopted, the QPID implementation will be brought into compliance with that standard.

4.3. Links

- The schema is checked into ???.

4.3.1. Design note for getting info in and out via JMX

???

4.4. Management Requirements

- Must operate from a formally defined management schema.
- Must natively use the AMQP protocol and its type system.
- Must support the following operations
  - SET operation on configurable (persistent) aspects of objects
  - GET operation on all aspects of objects
  - METHOD invocation on schema-defined object-specific methods
  - Distribution of unsolicited periodic updates of instrumentation data
    - Data updates shall carry an accurate sample timestamp for rate calculation
    - Updates shall carry object create/delete timestamps.
    - Transient objects shall be fully accounted for via updates. Note that short-lived transient objects may come and go within a single update interval. All of the information pertaining to such an object must be captured and transmitted.
  - Distribution of unsolicited event and/or alert indications (schema defined)
- Role-based access control at object, operation, and method granularity
- End-to-end encryption and signing of management content
- Schema must be self-describing so the management client need not have prior knowledge of the management model of the system under management.
- Must be extensible to support the management of objects beyond the QPID component set. This allows AMQP to be used as a general-purpose management protocol.

4.5. Definition of Terms

Table 6.6.

class	A type definition for a manageable object.
package	A grouping of class definitions that are related to a single software component. The package concept is used to extend the management schema beyond just the QPID software components.
object	Also "manageable object". An instantiation of a class. An object represents a physical or logical component in the core function of the system under management.
property	A typed member of a class which represents a configurable attribute of the class. In general, properties don't change frequently or may not change at all.
statistic	A typed member of a class which represents an instrumentation attribute of the class. Statistics are always read-only in nature and tend to change rapidly.
method	A member of a class which represents a callable procedure on an object of the class. Methods may have an arbitrary set of typed arguments and may supply a return code. Methods typically have side effects on the associated object.
event	A member of a class which represents the occurence of an event of interest within the system under management.
management broker	A software component built into the messaging broker that handles management traffic and distributes management data.
management agent	A software component that is separate from the messaging broker, connected to the management broker via an AMQP connection, which allows any software component to be managed remotely by QPID.

4.6. Operational Scenarios: Basic vs. Extended

The extensibility requirement introduces complexity to the management protocol that is unnecessary and undesirable for the user/developer that wishes only to manage QPID message brokers. For this reason, the protocol is partitioned into two parts: The *basic protocol*, which contains only the capability to manage a single broker; and the *extended protocol*, which provides the hooks for managing an extended set of components. A management console can be implemented using only the basic protocol if the extended capabilities are not needed.

4.7. Architectural Framework

???

4.8. The Management Exchange

The management exchange (called "qpid.management" currently) is a special type of exchange used for remote management access to the Qpid broker. The management exchange is an extension of the standard "Topic" exchange. It behaves like a topic exchange with the following exceptions:

1. When a queue is successfully bound to the exchange, a method is invoked on the broker's management agent to notify it of the presence of a new remote managment client.

2. When messages arrive at the exchange for routing, the exchange examines the message's routing key and if the key represents a management command or method, it routes it directly to the management agent rather than routing it to queues using the topic algorithm. The management exchange is used by the management agent to distribute unsolicited management data. Such data is classified by the routing key allowing management clients to register for only the data they need.

4.8.1. Routing Key Structure

As noted above, the structure of the binding and routing keys used on the management exchange is important to the function of the management architecture. The routing key of a management message determines:

1. The type of message (i.e. operation request or unsolicited update).

2. The class of the object that the message pertains to.

3. The specific operation or update type.

4. The namespace in which the class belongs. This allows for plug-in expansion of the management schema for manageable objects that are outside of the broker itself.

Placing this information in the routing key provides the ability to enforce access control at class, operation, and method granularity. It also separates the command structure from the content of the management message (i.e. element values) allowing the content to be encrypted and signed end-to-end while still allowing access control at the message-transport level. This means that special access control code need not be written for the management agent. There are two general types of routing/binding key:

- *Command* messages use the key: agent.<bank#> or broker

- *Unsolicited* keys have the structure: mgmt.<agent>.<type>.<package>.<class>.<severity> where

  - <agent> is the uuid of the originating management agent,

  - <type> is one of "schema", "prop", "stat", or "event",

  - <package> is the namespace in which the <class> name is valid, and

  - <class> is the name of the class as defined in the schema.

  - <severity> is relevant for events only. It is one of "critical", "error", "warning", or "info".

In both cases, the content of the message (i.e. method arguments, element values, etc.) is carried in the body segment of the message.

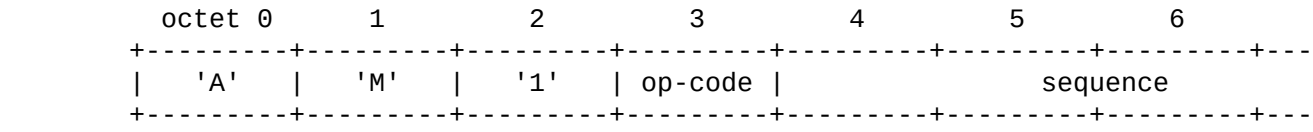
The <package> namespace allows this management framework to be extended with the addition of other software packages.

4.9. The Protocol

4.9.1. Protocol Header

The body segments of management messages are composed of sequences of binary-encoded data fields, in a manner consistent with the 0-10 version of the AMQP specification.

All management messages begin with a message header:



The first three octets contain the protocol *magic number* "AM1" which is used to identify the type and version of the message.

The *opcode* field identifies the operation represented by the message

4.9.2. Protocol Exchange Patterns

The following patterns are followed in the design of the protocol:

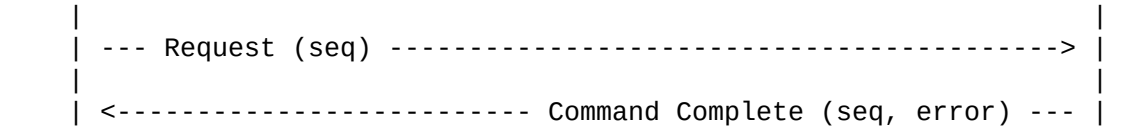
- Request-Response
- Query-Indication
- Unsolicited Indication

4.9.2.1. The Request-Response Pattern

In the request-response pattern, a requestor sends a *request* message to one of its peers. The peer then does one of two things: If the request can be successfully processed, a single *response* message is sent back to the requestor. This response contains the requested results and serves as the positive acknowledgement that the request was successfully completed.

If the request cannot be successfully completed, the peer sends a *command complete* message back to the requestor with an error code and error text describing what went wrong.

The sequence number in the *response* or *command complete* message is the same as the sequence number in the *request*.

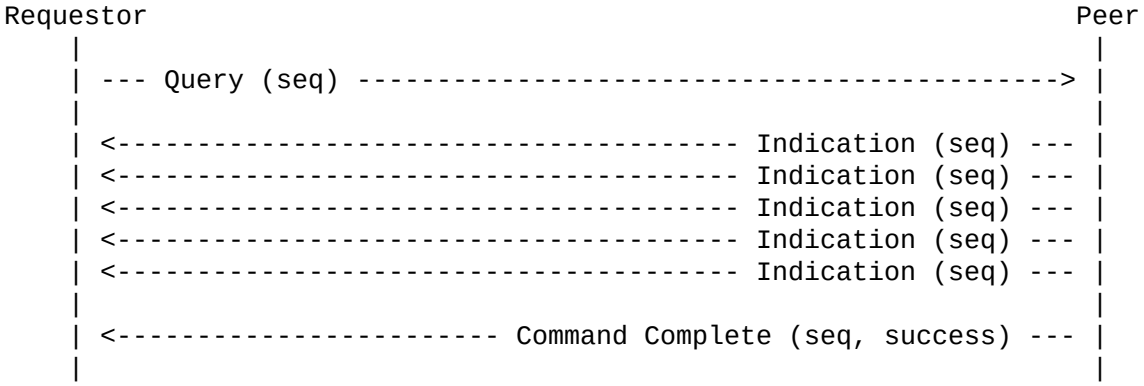


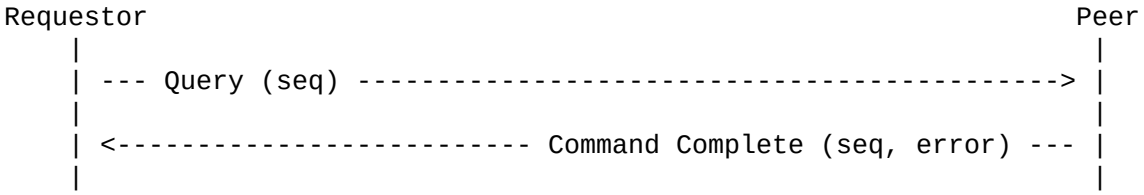
4.9.2.2. The Query-Indication Pattern

The query-indication pattern is used when there may be zero or more answers to a question. In this case, the requestor sends a *query* message to its peer. The peer processes the query, sending as many *indication* messages as needed back to the requestor (zero or more). Once the last *indication* has been sent, the peer then sends a *command complete* message with a success code indicating that the query is complete.

If there is an error in the *query*, the peer may reply with a *command complete* message containg an error code. In this case, no *indication* messages may be sent.

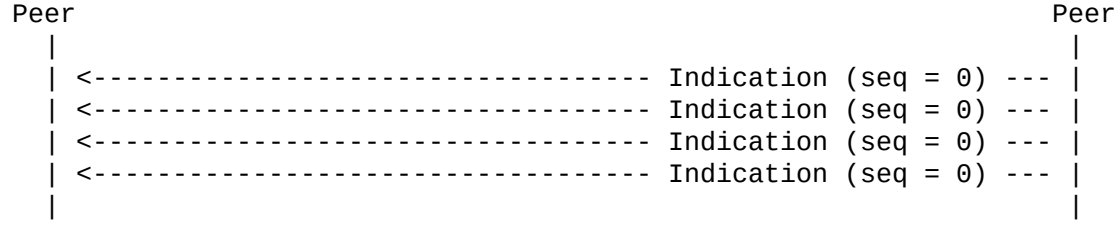
All *indication* and *command complete* messages shall have the same sequence number that appeared in the *query* message.





4.9.2.3. The Unsolicited-Indication Pattern

The unsolicited-indication pattern is used when one peer needs to send unsolicited information to another peer, or to broadcast information to multiple peers via a topic exchange. In this case, indication messages are sent with the sequence number field set to zero.

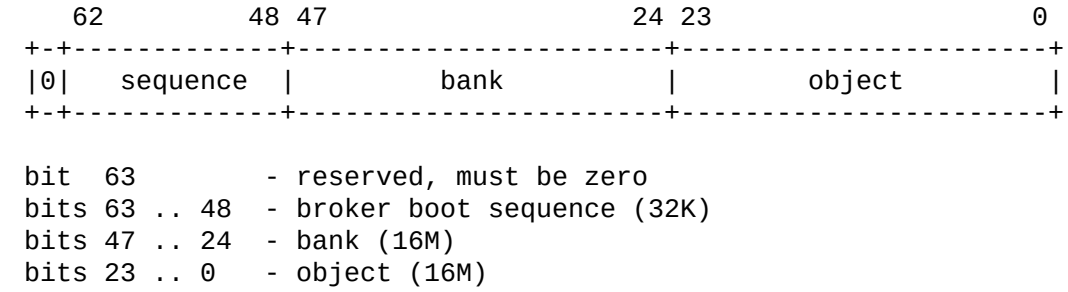


4.9.3. Object Identifiers

Manageable objects are tagged with a unique 64-bit object identifier. The object identifier space is owned and managed by the management broker. Objects managed by a single management broker shall have

unique object identifiers. Objects managed by separate management brokers may have the same object identifier.

If a management console is designed to manage multiple management brokers, it must use the broker identifier as well as the object identifier to ensure global uniqueness.



- For persistent IDs, boot-sequence is zero
- For non-persistent IDs, boot sequence is a constant number which increments each time the management broker is restarted.
- Bank number:
  - 0 - reserved
  - 1 - broker-persistent objects
  - 2..4 - store-persistent objects
  - > 4 - transient objects

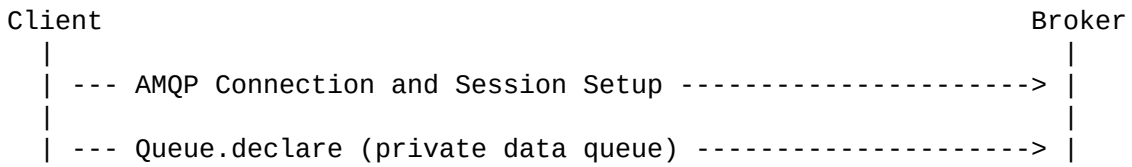
4.9.4. Establishing Communication Between Client and Agent

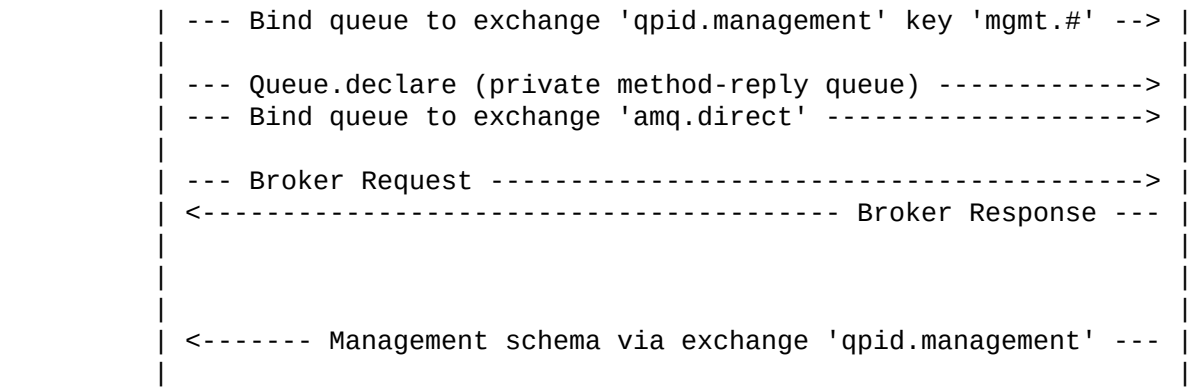
Communication is established between the management client and management agent using normal AMQP procedures. The client creates a connection to the broker and then establishes a session with its corresponding channel.

Two private queues are then declared (only one if method invocation is not needed). A management queue is declared and bound to the qpid.management exchange. If the binding key is "mgmt.#", all management-related messages sent to the exchange will be received by this client. A more specific binding key will result in a more restricted set of messages being received (see the section on Routing Key Structure below).

If methods are going to be invoked on managed objects, a second private queue must be declared so the client can receive method replies. This queue is bound to the amq.direct exchange using a routing key equal to the name of the queue.

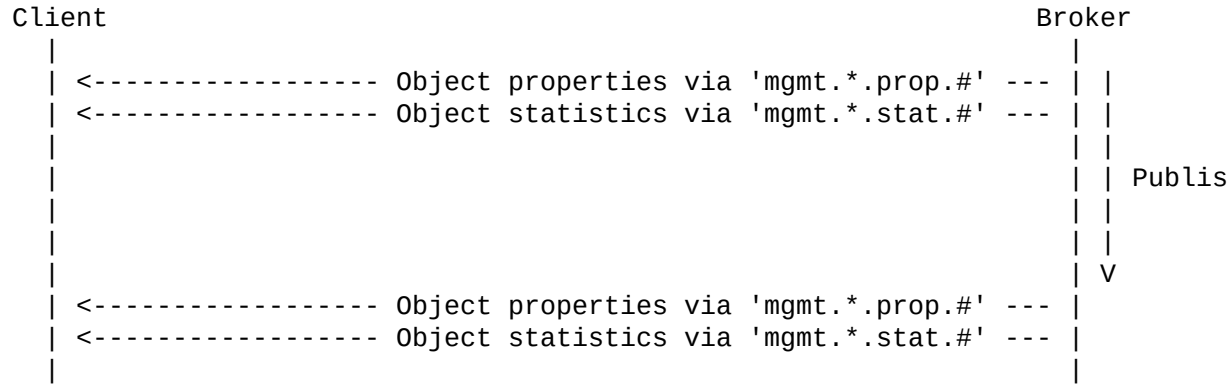
When a client successfully binds to the qpid.management exchange, the management agent schedules a schema broadcast to be sent to the exchange. The agent will publish, via the exchange, a description of the schema for all manageable objects in its control.





4.9.5. Broadcast of Configuration and Instrumentation Updates

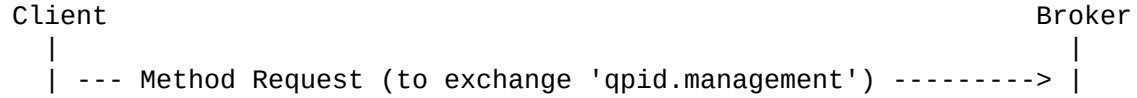
The management agent will periodically publish updates to the configuration and instrumentation of management objects under its control. Under normal circumstances, these updates are published only if they have changed since the last time they were published. Configuration updates are only published if configuration has changed and instrumentation updates are only published if instrumentation has changed. The exception to this rule is that after a management client binds to the qpid.management exchange, all configuration and instrumentation records are published as though they had changed whether or not they actually did.



4.9.6. Invoking a Method on a Managed Object

When the management client wishes to invoke a method on a managed object, it sends a method request message to the qpid.management exchange. The routing key contains the object class and method name (refer to Routing Key Structure below). The method request must have a header entry (reply-to) that contains the name of the method-reply queue so that the method response can be properly routed back to the requestor.

The method request contains a sequence number that is copied to the method reply. This number is opaque to the management agent and may be used by the management client to correlate the reply to the request. The asynchronous nature of requests and replies allows any number of methods to be in-flight at a time. Note that there is no guarantee that methods will be replied to in the order in which they were requested.



```
|                                                                  |
|                                                                  |
| <--------------- Method Reply (via exchange 'amq.direct') --- |
|                                                                  |
```

4.9.7. Messages for the Basic Scenario

The principals in a management exchange are the *management client* and the *management agent*. The management agent is integrated into the QPID broker and the management client is a remote entity. A management agent may be managed by zero or more management clients at any given time. Additionally, a management client may manage multiple management agents at the same time.

For authentication and access control, management relies on the mechanisms supplied by the AMQP protocol.

4.9.7.1. Basic Opcodes

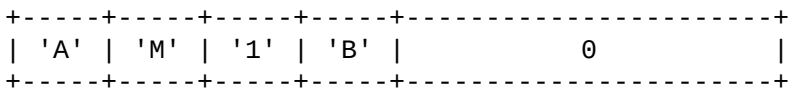
Table 6.7.

opcode	*message*	*description*
'B'	Broker Request	This message contains a broker request, sent from the management console to the broker to initiate a management session.
'b'	Broker Response	This message contains a broker response, sent from the broker in response to a broker request message.
'z'	Command Completion	This message is sent to indicate the completion of a request.
'Q'	Class Query	Class query messages are used by a management console to request a list of schema classes that are known by the management broker.
'q'	Class Indication	Sent by the management broker, a class indication notifies the peer of the existence of a schema class.
'S'	Schema Request	Schema request messages are used to request the full schema details for a class.
's'	Schema Response	Schema response message contain a full description of the schema for a class.
'h'	Heartbeat Indication	This message is published once per publish-interval. It can be used by a client to positively determine which objects did not change during the interval (since updates are not published for objects with no changes).

'c', 'i', 'g'	Content Indication	This message contains a content record. Content records contain the values of all properties or statistics in an object. Such records are broadcast on a periodic interval if 1) a change has been made in the value of one of the elements, or 2) if a new management client has bound a queue to the management exchange.
'G'	Get Query	Sent by a management console, a get query requests that the management broker provide content indications for all objects that match the query criteria.
'M'	Method Request	This message contains a method request.
'm'	Method Response	This message contains a method result.

4.9.7.2. Broker Request Message

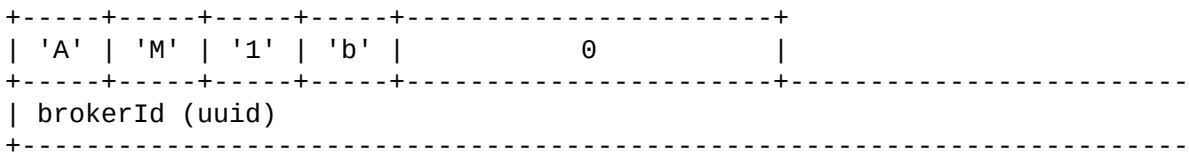
When a management client first establishes contact with the broker, it sends a Hello message to initiate the exchange.



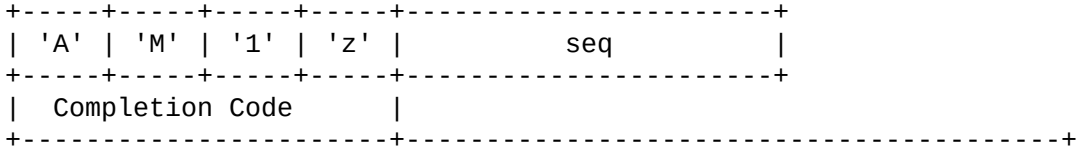
The Broker Request message has no payload.

4.9.7.3. Broker Response Message

When the broker receives a Broker Request message, it responds with a Broker Response message. This message contains an identifier unique to the broker.



4.9.7.4. Command Completion Message



```
|  Completion Text                                                |
+-----------------------------------------------------------------+
```

4.9.7.5. Class Query

```
+-----+-----+-----+-----+----------------------+
| 'A' | 'M' | '1' | 'Q' |          seq         |
+-----+-----+-----+-----+----------------------+----------+
|  package name (str8)                                     |
+----------------------------------------------------------+
```

4.9.7.6. Class Indication

```
+-----+-----+-----+-----+----------------------+
| 'A' | 'M' | '1' | 'q' |          seq         |
+-----+-----+-----+-----+----------------------+----------+
|  package name (str8)                                     |
+----------------------------------------------------------+
|  class name (str8)                                       |
+----------------------------------------------------------+
|  schema hash (bin128)                                    |
+----------------------------------------------------------+
```

4.9.7.7. Schema Request

```
+-----+-----+-----+-----+----------------------+
| 'A' | 'M' | '1' | 'S' |          seq         |
+-----+-----+-----+-----+----------------------+----------+
|               packageName (str8)                         |
+----------------------------------------------------------+
|               className (str8)                           |
+----------------------------------------------------------+
|               schema-hash (bin128)                       |
+----------------------------------------------------------+
```

4.9.7.8. Schema Response

```
+-----+-----+-----+-----+----------------------+
| 'A' | 'M' | '1' | 's' |          seq         |
+-----+-----+-----+-----+----------------------+----------+
|               packageName (str8)                         |
+----------------------------------------------------------+
|               className (str8)                           |
+----------------------------------------------------------+
|               schema-hash (bin128)                       |
+-----------+-----------+-----------+-----------+----------+
| propCnt   | statCnt   | methodCnt | eventCnt  |
+-----------+-----------+-----------+-----------+-------------------------
| propCnt property records
+-------------------------------------------------------------------------
```

```
| statCnt statistic records
+-------------------------------------------------------------------------
| methodCnt method records
+-------------------------------------------------------------------------
| eventCnt event records
+-------------------------------------------------------------------------
```

Each *property* record is an AMQP map with the following fields. Optional fields may optionally be omitted from the map.

Table 6.8.

field name	*optional*	*description*
name	no	Name of the property
type	no	Type code for the property
access	no	Access code for the property
index	no	1 = index element, 0 = not an index element
optional	no	1 = optional element (may be not present), 0 = mandatory (always present)
unit	yes	Units for numeric values (i.e. seconds, bytes, etc.)
min	yes	Minimum value for numerics
max	yes	Maximum value for numerics
maxlen	yes	Maximum length for strings
desc	yes	Description of the property

Each *statistic* record is an AMQP map with the following fields:

Table 6.9.

field name	*optional*	*description*
name	no	Name of the statistic
type	no	Type code for the statistic
unit	yes	Units for numeric values (i.e. seconds, bytes, etc.)
desc	yes	Description of the statistic

method and *event* records contain a main map that describes the method or header followed by zero or more maps describing arguments. The main map contains the following fields:

Table 6.10.

field name	*optional*	*description*
name	no	Name of the method or event
argCount	no	Number of argument records to follow

desc	yes	Description of the method or event

Argument maps contain the following fields:

Table 6.11.

field name	*method*	*event*	*optional*	*description*
name	yes	yes	no	Argument name
type	yes	yes	no	Type code for the argument
dir	yes	no	yes	Direction code for method arguments
unit	yes	yes	yes	Units for numeric values (i.e. seconds, bytes, etc.)
min	yes	no	yes	Minimum value for numerics
max	yes	no	yes	Maximum value for numerics
maxlen	yes	no	yes	Maximum length for strings
desc	yes	yes	yes	Description of the argument
default	yes	no	yes	Default value for the argument

type codes are numerics with the following values:

Table 6.12.

value	*type*
1	uint8
2	uint16
3	uint32
4	uint64
6	str8
7	str16
8	absTime(uint64)
9	deltaTime(uint64)
10	objectReference(uint64)
11	boolean(uint8)
12	float
13	double
14	uuid

15	map
16	int8
17	int16
18	int32
19	int64

access codes are numerics with the following values:

Table 6.13.

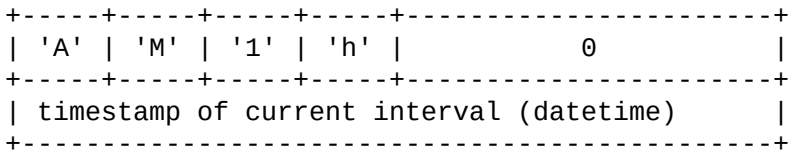
value	*access*
1	Read-Create access
2	Read-Write access
3	Read-Only access

direction codes are numerics with the following values:

Table 6.14.

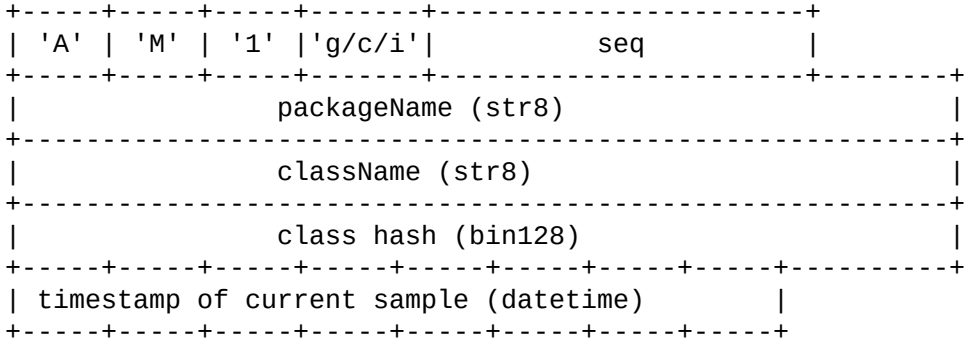
value	*direction*
1	Input (from client to broker)
2	Output (from broker to client)
3	IO (bidirectional)

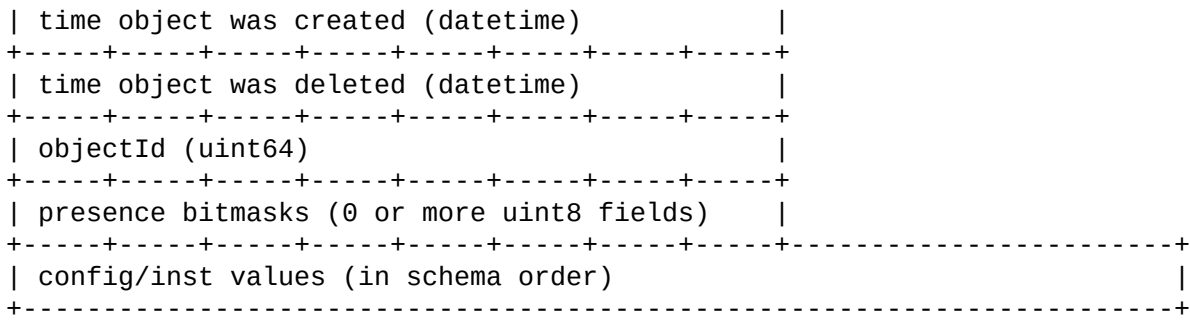
4.9.7.9. Heartbeat Indication



4.9.7.10. Configuration and Instrumentation Content Messages

Content messages are published when changes are made to the values of properties or statistics or when new management clients bind a queue to the management exchange.





All timestamps are uint64 values representing nanoseconds since the epoch (January 1, 1970). The objectId is a uint64 value that uniquely identifies this object instance.

If any of the properties in the object are defined as optional, there will be 1 or more "presence bitmask" octets. There are as many octets as are needed to provide one bit per optional property. The bits are assigned to the optional properties in schema order (first octet first, lowest order bit first).

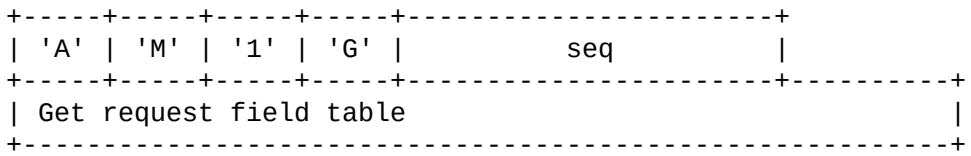
For example: If there are two optional properties in the schema called "option1" and "option2" (defined in that order), there will be one presence bitmask octet and the bits will be assigned as bit 0 controls option1 and bit 1 controls option2.

If the bit for a particular optional property is set (1), the property will be encoded normally in the "values" portion of the message. If the bit is clear (0), the property will be omitted from the list of encoded values and will be considered "NULL" or "not present".

The element values are encoded by their type into the message in the order in which they appeared in the schema message.

4.9.7.11. Get Query Message

A Get Request may be sent by the management console to cause a management agent to immediately send content information for objects of a class.



The content of a get request is a field table that specifies what objects are being requested. Most of the fields are optional and are available for use in more extensive deployments.

Table 6.15.

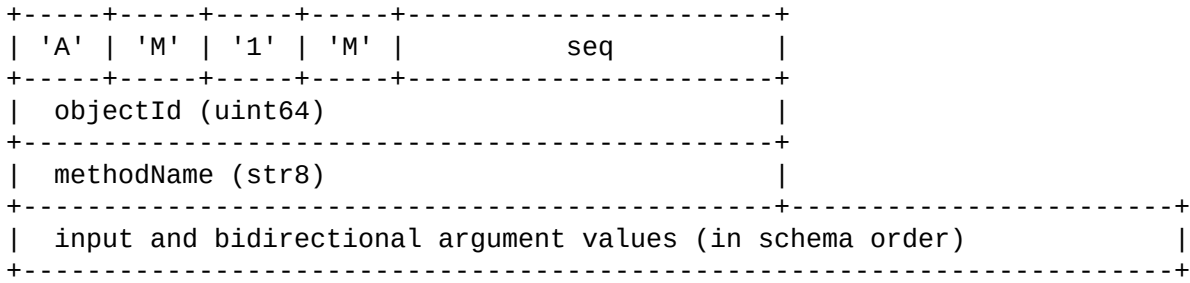
Field Key	*Mandatory*	*Type*	*Description*
"_class"	yes	short-string	The name of the class of objects being requested.
"_package"	no	short-string	The name of the extension package the class belongs to. If omitted, the package defaults to "qpid" for

			access to objects in the connected broker.
"_agent"	no	uuid	The management agent that is the target of the request. If omitted, agent defaults to the connected broker.

When the management agent receives a get request, it sends content messages describing the requested objects. Once the last content message is sent, it then sends a Command Completion message with the same sequence number supplied in the request to indicate to the requestor that there are no more messages coming.

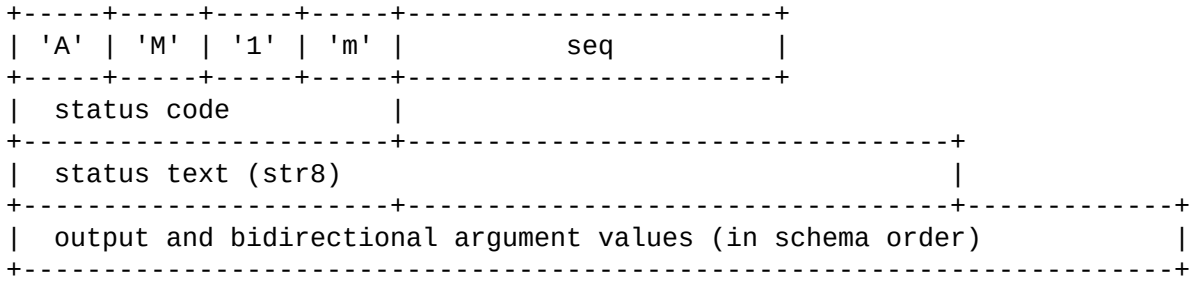
4.9.7.12. Method Request

Method request messages have the following structure. The sequence number is opaque to the management agent. It is returned unchanged in the method reply so the calling client can correctly associate the reply to the request. The objectId is the unique ID of the object on which the method is to be executed.



4.9.7.13. Method Response

Method reply messages have the following structure. The sequence number is identical to that supplied in the method request. The status code (and text) indicate whether or not the method was successful and if not, what the error was. Output and bidirectional arguments are only included if the status code was 0 (STATUS_OK).



status code values are:

Table 6.16.

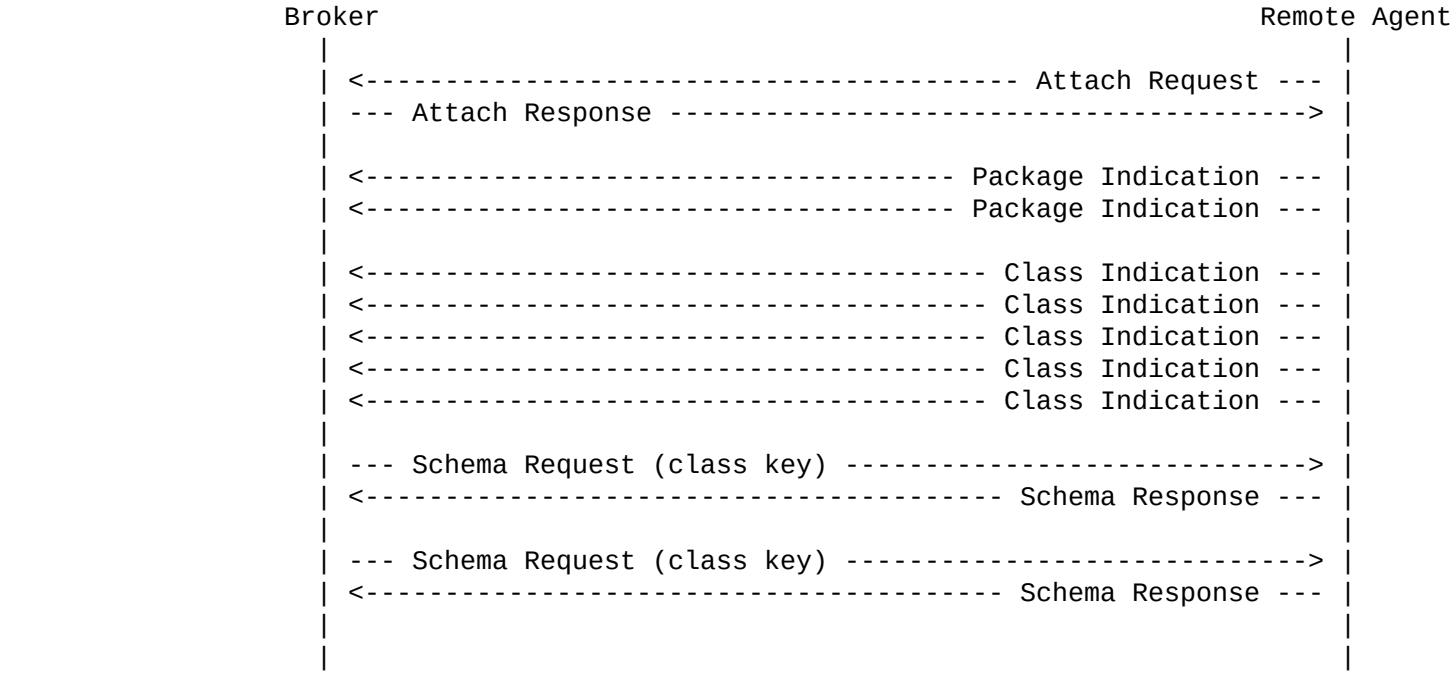
value	*description*
0	STATUS_OK - successful completion

1	STATUS_UNKNOWN_OBJECT - objectId not found in the agent
2	STATUS_UNKNOWN_METHOD - method is not known by the object type
3	STATUS_NOT_IMPLEMENTED - method is not currently implemented

4.9.8. Messages for Extended Scenario

4.9.8.1. Extended Management Protocol

Qpid supports management extensions that allow the management broker to be a central point for the management of multiple external entities with their own management schemas.



4.9.8.2. Extended Opcodes

Table 6.17.

opcode	*message*	*description*
'P'	Package Query	This message contains a schema package query request, requesting that the broker dump the list of known packages
'p'	Package Indication	This message contains a schema package indication, identifying a package known by the broker
'A'	Agent Attach Request	This message is sent by a remote agent when it wishes to attach to a management broker

'a'	Agent Attach Response	The management broker sends this response if an attaching remote agent is permitted to join
'x'	Console Added Indication	This message is sent to all remote agents by the management broker when a new console binds to the management exchange

4.9.8.3. Package Query

```
+-----+-----+-----+-----+-----------------------+
| 'A' | 'M' | '1' | 'P' |          seq          |
+-----+-----+-----+-----+-----------------------+
```

4.9.8.4. Package Indication

```
+-----+-----+-----+-----+-----------------------+
| 'A' | 'M' | '1' | 'p' |          seq          |
+-----+-----+-----+-----+-----------------------+----------+
|  package name (str8)                                     |
+----------------------------------------------------------+
```

4.9.8.5. Attach Request

```
+-----+-----+-----+-----+-----------------------+
| 'A' | 'M' | '1' | 'A' |          seq          |
+-----+-----+-----+-----+-----------------------+----------+
|  label (str8)                                            |
+------------------------+---------------------------------+
|  system-id (uuid)                                        |
+------------------------+---------------------------------+
|  requested objId bank  |
+------------------------+
```

4.9.8.6. Attach Response (success)

```
+-----+-----+-----+-----+-----------------------+
| 'A' | 'M' | '1' | 'a' |          seq          |
+-----+-----+-----+-----+-----------------------+
|  assigned broker bank  |
+------------------------+
|  assigned objId bank   |
+------------------------+
```

4.9.8.7. Console Added Indication

```
+-----+-----+-----+-----+-----------------------+
| 'A' | 'M' | '1' | 'x' |          seq          |
+-----+-----+-----+-----+-----------------------+
```

5. QMF Python Console Tutorial

5.1. Prerequisite - Install Qpid Messaging

QMF uses AMQP Messaging (QPid) as its means of communication. To use QMF, Qpid messaging must be installed somewhere in the network. Qpid can be downloaded as source from Apache, is packaged with a number of Linux distributions, and can be purchased from commercial vendors that use Qpid. Please see http://qpid.apache.orgfor information as to where to get Qpid Messaging.

Qpid Messaging includes a message broker (qpidd) which typically runs as a daemon on a system. It also includes client bindings in various programming languages. The Python-language client library includes the QMF console libraries needed for this tutorial.

Please note that Qpid Messaging has two broker implementations. One is implemented in C++ and the other in Java. At press time, QMF is supported only by the C++ broker.

If the goal is to get the tutorial examples up and running as quickly as possible, all of the Qpid components can be installed on a single system (even a laptop). For more realistic deployments, the broker can be deployed on a server and the client/QMF libraries installed on other systems.

5.2. Synchronous Console Operations

The Python console API for QMF can be used in a synchronous style, an asynchronous style, or a combination of both. Synchronous operations are conceptually simple and are well suited for user-interactive tasks. All operations are performed in the context of a Python function call. If communication over the message bus is required to complete an operation, the function call blocks and waits for the expected result (or timeout failure) before returning control to the caller.

5.2.1. Creating a QMF Console Session and Attaching to a Broker

For the purposes of this tutorial, code examples will be shown as they are entered in an interactive python session.

```
$ python
Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)
[GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We will begin by importing the required libraries. If the Python client is properly installed, these libraries will be found normally by the Python interpreter.

```
>>> from qmf.console import Session
```

We must now create a *Session* object to manage this QMF console session.

```
>>> sess = Session()
```

If no arguments are supplied to the creation of *Session*, it defaults to synchronous-only operation. It also defaults to user-management of connections. More on this in a moment.

We will now establish a connection to the messaging broker. If the broker daemon is running on the local host, simply use the following:

```
>>> broker = sess.addBroker()
```

If the messaging broker is on a remote host, supply the URL to the broker in the *addBroker* function call. Here's how to connect to a local broker using the URL.

```
>>> broker = sess.addBroker("amqp://localhost")
```

The call to *addBroker* is synchronous and will return only after the connection has been successfully established or has failed. If a failure occurs, *addBroker* will raise an exception that can be handled by the console script.

```
>>> try:
...   broker = sess.addBroker("amqp://localhost:1000")
... except:
...   print "Connection Failed"
...
Connection Failed
>>>
```

This operation fails because there is no Qpid Messaging broker listening on port 1000 (the default port for qpidd is 5672).

If preferred, the QMF session can manage the connection for you. In this case, *addBroker* returns immediately and the session attempts to establish the connection in the background. This will be covered in detail in the section on asynchronous operations.

5.2.2. Accessing Managed Objects

The Python console API provides access to remotely managed objects via a *proxy* model. The API gives the client an object that serves as a proxy representing the "real" object being managed on the agent application. Operations performed on the proxy result in the same operations on the real object.

The following examples assume prior knowledge of the kinds of objects that are actually available to be managed. There is a section later in this tutorial that describes how to discover what is manageable on the QMF bus.

Proxy objects are obtained by calling the *Session.getObjects* function.

To illustrate, we'll get a list of objects representing queues in the message broker itself.

```
>>> queues = sess.getObjects(_class="queue", _package="org.apache.qpid.broker")
```

queues is an array of proxy objects representing real queues on the message broker. A proxy object can be printed to display a description of the object.

```
>>> for q in queues:
...   print q
...
org.apache.qpid.broker:queue[0-1537-1-0-58] 0-0-1-0-1152921504606846979:reply-loca
org.apache.qpid.broker:queue[0-1537-1-0-61] 0-0-1-0-1152921504606846979:topic-loca
>>>
```

5.2.2.1. Viewing Properties and Statistics of an Object

Let us now focus our attention on one of the queue objects.

```
>>> queue = queues[0]
```

The attributes of an object are partitioned into *properties* and *statistics*. Though the distinction is somewhat arbitrary, *properties* tend to be fairly static and may also be large and *statistics* tend to change rapidly and are relatively small (counters, etc.).

There are two ways to view the properties of an object. An array of properties can be obtained using the *getProperties* function:

```
>>> props = queue.getProperties()
>>> for prop in props:
...   print prop
...
(vhostRef, 0-0-1-0-1152921504606846979)
(name, u'reply-localhost.localdomain.32004')
(durable, False)
(autoDelete, True)
(exclusive, True)
(arguments, {})
>>>
```

The *getProperties* function returns an array of tuples. Each tuple consists of the property descriptor and the property value.

A more convenient way to access properties is by using the attribute of the proxy object directly:

```
>>> queue.autoDelete
```

```
True
>>> queue.name
u'reply-localhost.localdomain.32004'
>>>
```

Statistics are accessed in the same way:

```
>>> stats = queue.getStatistics()
>>> for stat in stats:
...   print stat
...
(msgTotalEnqueues, 53)
(msgTotalDequeues, 53)
(msgTxnEnqueues, 0)
(msgTxnDequeues, 0)
(msgPersistEnqueues, 0)
(msgPersistDequeues, 0)
(msgDepth, 0)
(byteDepth, 0)
(byteTotalEnqueues, 19116)
(byteTotalDequeues, 19116)
(byteTxnEnqueues, 0)
(byteTxnDequeues, 0)
(bytePersistEnqueues, 0)
(bytePersistDequeues, 0)
(consumerCount, 1)
(consumerCountHigh, 1)
(consumerCountLow, 1)
(bindingCount, 2)
(bindingCountHigh, 2)
(bindingCountLow, 2)
(unackedMessages, 0)
(unackedMessagesHigh, 0)
(unackedMessagesLow, 0)
(messageLatencySamples, 0)
(messageLatencyMin, 0)
(messageLatencyMax, 0)
(messageLatencyAverage, 0)
>>>
```

or alternatively:

```
>>> queue.byteTotalEnqueues
19116
>>>
```

The proxy objects do not automatically track changes that occur on the real objects. For example, if the real queue enqueues more bytes, viewing the *byteTotalEnqueues* statistic will show the same number as it did the first time. To get updated data on a proxy object, use the *update* function call:

```
>>> queue.update()
>>> queue.byteTotalEnqueues
```

```
19783
>>>
```

Be Advised

The *update* method was added after the M4 release of Qpid/Qmf. It may not be available in your distribution.

5.2.2.2. Invoking Methods on an Object

Up to this point, we have used the QMF Console API to find managed objects and view their attributes, a read-only activity. The next topic to illustrate is how to invoke a method on a managed object. Methods allow consoles to control the managed agents by either triggering a one-time action or by changing the values of attributes in an object.

First, we'll cover some background information about methods. A *QMF object class* (of which a *QMF object* is an instance), may have zero or more methods. To obtain a list of methods available for an object, use the *getMethods* function.

```
>>> methodList = queue.getMethods()
```

getMethods returns an array of method descriptors (of type qmf.console.SchemaMethod). To get a summary of a method, you can simply print it. The *_repr_* function returns a string that looks like a function prototype.

```
>>> print methodList
[purge(request)]
>>>
```

For the purposes of illustration, we'll use a more interesting method available on the *broker* object which represents the connected Qpid message broker.

```
>>> br = sess.getObjects(_class="broker", _package="org.apache.qpid.broker")[0]
>>> mlist = br.getMethods()
>>> for m in mlist:
...   print m
...
echo(sequence, body)
connect(host, port, durable, authMechanism, username, password, transport)
queueMoveMessages(srcQueue, destQueue, qty)
>>>
```

We have just learned that the *broker* object has three methods: *echo*, *connect*, and *queueMoveMessages*. We'll use the *echo* method to "ping" the broker.

```
>>> result = br.echo(1, "Message Body")
>>> print result
OK (0) - {'body': u'Message Body', 'sequence': 1}
>>> print result.status
0
>>> print result.text
OK
```

```
>>> print result.outArgs
{'body': u'Message Body', 'sequence': 1}
>>>
```

In the above example, we have invoked the *echo* method on the instance of the broker designated by the proxy "br" with a sequence argument of 1 and a body argument of "Message Body". The result indicates success and contains the output arguments (in this case copies of the input arguments).

To be more precise... Calling *echo* on the proxy causes the input arguments to be marshalled and sent to the remote agent where the method is executed. Once the method execution completes, the output arguments are marshalled and sent back to the console to be stored in the method result.

You are probably wondering how you are supposed to know what types the arguments are and which arguments are input, which are output, or which are both. This will be addressed later in the "Discovering what Kinds of Objects are Available" section.

5.3. Asynchronous Console Operations

QMF is built on top of a middleware messaging layer (Qpid Messaging). Because of this, QMF can use some communication patterns that are difficult to implement using network transports like UDP, TCP, or SSL. One of these patterns is called the *Publication and Subscription* pattern (pub-sub for short). In the pub-sub pattern, data sources *publish* information without a particular destination in mind. Data sinks (destinations) *subscribe* using a set of criteria that describes what kind of data they are interested in receiving. Data published by a source may be received by zero, one, or many subscribers.

QMF uses the pub-sub pattern to distribute events, object creation and deletion, and changes to properties and statistics. A console application using the QMF Console API can receive these asynchronous and unsolicited events and updates. This is useful for applications that store and analyze events and/or statistics. It is also useful for applications that react to certain events or conditions.

Note that console applications may always use the synchronous mechanisms.

5.3.1. Creating a Console Class to Receive Asynchronous Data

Asynchronous API operation occurs when the console application supplies a *Console* object to the session manager. The *Console* object (which overrides the *qmf.console.Console* class) handles all asynchronously arriving data. The *Console* class has the following methods. Any number of these methods may be overridden by the console application. Any method that is not overridden defaults to a null handler which takes no action when invoked.

Table 6.18. QMF Python Console Class Methods

Method	Arguments	Invoked when...
brokerConnected	broker	a connection to a broker is established
brokerDisconnected	broker	a connection to a broker is lost
newPackage	name	a new package is seen on the QMF bus
newClass	kind, classKey	a new class (event or object) is seen on the QMF bus
newAgent	agent	a new agent appears on the QMF bus

delAgent	agent	an agent disconnects from the QMF bus
objectProps	broker, object	the properties of an object are published
objectStats	broker, object	the statistics of an object are published
event	broker, event	an event is published
heartbeat	agent, timestamp	a heartbeat is published by an agent
brokerInfo	broker	information about a connected broker is available to be queried
methodResponse	broker, seq, response	the result of an asynchronous method call is received

Supplied with the API is a class called *DebugConsole*. This is a test *Console* instance that overrides all of the methods such that arriving asynchronous data is printed to the screen. This can be used to see all of the arriving asynchronous data.

5.3.2. Receiving Events

We'll start the example from the beginning to illustrate the reception and handling of events. In this example, we will create a *Console* class that handles broker-connect, broker-disconnect, and event messages. We will also allow the session manager to manage the broker connection for us.

Begin by importing the necessary classes:

```
>>> from qmf.console import Session, Console
```

Now, create a subclass of *Console* that handles the three message types:

```
>>> class EventConsole(Console):
...   def brokerConnected(self, broker):
...     print "brokerConnected:", broker
...   def brokerDisconnected(self, broker):
...     print "brokerDisconnected:", broker
...   def event(self, broker, event):
...     print "event:", event
...
>>>
```

Make an instance of the new class:

```
>>> myConsole = EventConsole()
```

Create a *Session* class using the console instance. In addition, we shall request that the session manager do the connection management for us. Notice also that we are requesting that the session manager not receive objects or heartbeats. Since this example is concerned only with events, we can optimize the use of the messaging bus by telling the session manager not to subscribe for object updates or heartbeats.

```
>>> sess = Session(myConsole, manageConnections=True, rcvObjects=False, rcvHeartbe
>>> broker = sess.addBroker()
>>>
```

Once the broker is added, we will begin to receive asynchronous events (assuming there is a functioning broker available to connect to).

```
brokerConnected: Broker connected at: localhost:5672
event: Thu Jan 29 19:53:19 2009 INFO  org.apache.qpid.broker:bind broker=localhost
```

5.3.3. Receiving Objects

To illustrate asynchronous handling of objects, a small console program is supplied. The entire program is shown below for convenience. We will then go through it part-by-part to explain its design.

This console program receives object updates and displays a set of statistics as they change. It focuses on broker queue objects.

```
# Import needed classes
from qmf.console import Session, Console
from time        import sleep

# Declare a dictionary to map object-ids to queue names
queueMap = {}

# Customize the Console class to receive object updates.
class MyConsole(Console):

  # Handle property updates
  def objectProps(self, broker, record):

    # Verify that we have received a queue object.  Exit otherwise.
    classKey = record.getClassKey()
    if classKey.getClassName() != "queue":
      return

    # If this object has not been seen before, create a new mapping from objectID
    oid = record.getObjectId()
    if oid not in queueMap:
      queueMap[oid] = record.name

  # Handle statistic updates
  def objectStats(self, broker, record):

    # Ignore updates for objects that are not in the map
    oid = record.getObjectId()
    if oid not in queueMap:
      return

    # Print the queue name and some statistics
    print "%s: enqueues=%d dequeues=%d" % (queueMap[oid], record.msgTotalEnqueues,
```

```
    # if the delete-time is non-zero, this object has been deleted.  Remove it fro
    if record.getTimestamps()[2] > 0:
      queueMap.pop(oid)

# Create an instance of the QMF session manager.  Set userBindings to True to allo
# this program to choose which objects classes it is interested in.
sess = Session(MyConsole(), manageConnections=True, rcvEvents=False, userBindings=

# Register to receive updates for broker:queue objects.
sess.bindClass("org.apache.qpid.broker", "queue")
broker = sess.addBroker()

# Suspend processing while the asynchronous operations proceed.
try:
  while True:
    sleep(1)
except:
  pass

# Disconnect the broker before exiting.
sess.delBroker(broker)
```

Before going through the code in detail, it is important to understand the differences between synchronous object access and asynchronous object access. When objects are obtained synchronously (using the *getObjects* function), the resulting proxy contains all of the object's attributes, both properties and statistics. When object data is published asynchronously, the properties and statistics are sent separately and only when the session first connects or when the content changes.

The script wishes to print the queue name with the updated statistics, but the queue name is only present with the properties. For this reason, the program needs to keep some state to correlate property updates with their corresponding statistic updates. This can be done using the *ObjectId* that uniquely identifies the object.

```
    # If this object has not been seen before, create a new mapping from objectID
    oid = record.getObjectId()
    if oid not in queueMap:
      queueMap[oid] = record.name
```

The above code fragment gets the object ID from the proxy and checks to see if it is in the map (i.e. has been seen before). If it is not in the map, a new map entry is inserted mapping the object ID to the queue's name.

```
    # if the delete-time is non-zero, this object has been deleted.  Remove it fro
    if record.getTimestamps()[2] > 0:
      queueMap.pop(oid)
```

This code fragment detects the deletion of a managed object. After reporting the statistics, it checks the timestamps of the proxy. *getTimestamps* returns a list of timestamps in the order:

- *Current* - The timestamp of the sending of this update.
- *Create* - The time of the object's creation
- *Delete* - The time of the object's deletion (or zero if not deleted)

This code structure is useful for getting information about very-short-lived objects. It is possible that an object will be created, used, and deleted within an update interval. In this case, the property update will arrive first, followed by the statistic update. Both will indicate that the object has been deleted but a full accounting of the object's existence and final state is reported.

```
# Create an instance of the QMF session manager.  Set userBindings to True to allo
# this program to choose which objects classes it is interested in.
sess = Session(MyConsole(), manageConnections=True, rcvEvents=False, userBindings=

# Register to receive updates for broker:queue objects.
sess.bindClass("org.apache.qpid.broker", "queue")
```

The above code is illustrative of the way a console application can tune its use of the QMF bus. Note that *rcvEvents* is set to False. This prevents the reception of events. Note also the use of *userBindings=True* and the call to *sess.bindClass*. If *userBindings* is set to False (its default), the session will receive object updates for all classes of object. In the case above, the application is only interested in broker:queue objects and reduces its bus bandwidth usage by requesting updates to only that class. *bindClass* may be called as many times as desired to add classes to the list of subscribed classes.

5.3.4. Asynchronous Method Calls and Method Timeouts

Method calls can also be invoked asynchronously. This is useful if a large number of calls needs to be made in a short time because the console application will not need to wait for the complete round-trip delay for each call.

Method calls are synchronous by default. They can be made asynchronous by adding the keyword-argument *_async=True* to the method call.

In a synchronous method call, the return value is the method result. When a method is called asynchronously, the return value is a sequence number that can be used to correlate the eventual result to the request. This sequence number is passed as an argument to the *methodResponse* function in the *Console* interface.

It is important to realize that the *methodResponse* function may be invoked before the asynchronous call returns. Make sure your code is written to handle this possibility.

5.4. Discovering what Kinds of Objects are Available

Part III. AMQP Messaging Broker (Implemented in Java)

Qpid provides two AMQP messaging brokers:

- Implemented in C++ - high performance, low latency, and RDMA support.
- Implemented in Java - Fully JMS compliant, runs on any Java platform.

Both AMQP messaging brokers support clients in multiple languages, as long as the messaging client and the messaging broker use the same version of AMQP. See ??? to see which messaging clients work with each broker.

This section contains information specific to the broker that is implemented in Java.

Chapter 7. General User Guides

1. Java Broker Feature Guide

1.1. The Qpid pure Java broker currently supports the following features:

- All features required by the Sun JMS 1.1 specification, fully tested
- Transaction support
- Persistence using a pluggable layer
- Pluggable security using SASL
- Management using JMX and an Eclipse Management Console application
- High performance header-based routing for messages
- Message Priorities
- Configurable logging and log archiving
- Threshold alerting
- ACLs
- Extensively tested on each release, including performance & reliability testing
- Automatic client failover using configurable connection properties
- Durable Queues/Subscriptions

1.1.1. Upcoming features:

- Flow To Disk
- IP Whitelist
- AMQP 0-10 Support (for interoperability)

2. Qpid Java FAQ

2.1. Purpose

Here are a list of commonly asked questions and answers. Click on the the bolded questions for the answer to unfold. If you have any questions which are not on this list, please email our qpid-user list.

2.1.1. What is Qpid ?

The java implementation of Qpid is a pure Java message broker that implements the AMQP protocol. Essentially, Qpid is a robust, performant middleware component that can handle your messaging traffic.

It currently supports the following features:

- High performance header-based routing for messages
- All features required by the JMS 1.1 specification. Qpid passes all tests in the Sun JMS compliance test suite
- Transaction support
- Persistence using the high performance Berkeley DB Java Edition. The persistence layer is also pluggable should an alternative implementation be required. The BDB store is available from the Section 3, “QpidComponents.org” page
- Pluggable security using SASL. Any Java SASL provider can be used
- Management using JMX and a custom management console built using Eclipse RCP
- Naturally, interoperability with other clients including the Qpid .NET, Python, Ruby and C++ implementations

2.1.2. Why am I getting a ConfigurationException at broker startup ?

2.1.2.1. InvocationTargetException

If you get a java.lang.reflect.InvocationTargetException on startup, wrapped as ConfigurationException like this:

```
Error configuring message broker: org.apache.commons.configuration.ConfigurationEx
2008-09-26 15:14:56,529 ERROR [main] server.Main (Main.java:206) - Error configuri
org.apache.commons.configuration.ConfigurationException: java.lang.reflect.Invocat
at org.apache.qpid.server.security.auth.database.ConfigurationFilePrincipalDatabas
at org.apache.qpid.server.security.auth.database.ConfigurationFilePrincipalDatabas
at org.apache.qpid.server.security.auth.database.ConfigurationFilePrincipalDatabas
at org.apache.qpid.server.registry.ConfigurationFileApplicationRegistry.initialise
at org.apache.qpid.server.registry.ApplicationRegistry.initialise(ApplicationRegis
at org.apache.qpid.server.registry.ApplicationRegistry.initialise(ApplicationRegis
at org.apache.qpid.server.Main.startup(Main.java:260)
at org.apache.qpid.server.Main.execute(Main.java:196)
at org.apache.qpid.server.Main.<init>(Main.java:96)
at org.apache.qpid.server.Main.main(Main.java:454)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
at java.lang.reflect.Method.invoke(Method.java:597)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:90)
Caused by: java.lang.reflect.InvocationTargetException
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
at java.lang.reflect.Method.invoke(Method.java:597)
at org.apache.qpid.server.security.auth.database.ConfigurationFilePrincipalDatabas
```

.. then it means you have a missing password file.

You need to create a password file for your deployment and update your config.xml to reflect the location of the password file for your instance.

The config.xml can be a little confusing in terms of element names and file names for passwords.

To do this, you need to edit the passwordDir element for the broker, which may have a comment to that effect:

```
<passwordDir><!-- Change to the location --></passwordDir>
```

The file should be named passwd by default but if you want to you can change this by editing this element:

```
<value>${passwordDir}/passwd</value>
```

2.1.2.2. Cannot locate configuration source null/virtualhosts.xml

If you get this message, wrapped inside a ConfigurationException then you've come across a known issue, see JIRA ???

The work around is to use a qualified path as the parameter value for your -c option, rather than (as you migth be) starting the broker from your installed etc directory. Even going up one level and using a path relative to your Â£QPID_HOME directory would sort this e.g qpid-server -c ./etc/myconfig.xml

2.1.3. How do I run the Qpid broker ?

The broker comes with a script for unix/linux/cygwin called qpid-server, which can be found in the bin directory of the installed package. This command can be executed without any paramters and will then use the default configuration file provided on install.

For the Windows OS, please use qpid-server.bat.

There's no need to set your classpath for QPID as the scripts take care of that by adding jar's with classpath defining manifest files to your classpath.

For more information on running the broker please see our Chapter 3, *Getting Started* page.

2.1.4. How can I create a connection using a URL ?

Please see the ??? documentation.

2.1.5. How do I represent a JMS Destination string with QPID ?

2.1.5.1. Queues

A queue can be created in QPID using the following URL format.

direct://amq.direct/<Destination>/<Queue Name>

For example: direct://amq.direct/<Destination>/simpleQueue

Queue names may consist of any mixture of digits, letters, and underscores.

The Section 1.3, “ Binding URL Format ” is described in more detail on it's own page.

2.1.5.2. Topics

A topic can be created in QPID using the following URL format.

topic://amq.topic/<Topic Subscription>/

The topic subscription may only contain the letters A-Z and a-z and digits 0-9.

The topic subscription is formed from a series of words that may only contain the letters A-Z and a-z and digits 0-9. The words are delimited by dots. Each dot represents a new level.

For example: stocks.nyse.ibm

Wildcards can be used on subscription with the following meaning.

- match a single level # match zero or more levels

For example: With two clients 1 - stocks.*.ibm 2 - stocks.#.ibm

Publishing stocks.nyse.ibm will be received by both clients but stocks.ibm and stocks.world.us.ibm will only be received by client 2.

The topic currently does not support wild cards.

2.1.6. How do I connect to the broker using JNDI ?

see ???

2.1.7. I'm using Spring and Weblogic - can you help me with the configuration for moving over to Qpid ?

Here is a donated Spring configuration file appContext.zip [http://qpid.apache.org/qpid-java-faq.data/appContext.zip] which shows the config for Qpid side by side with Weblogic. HtH !

2.1.8. How do I configure the logging level for Qpid ?

The system property

```
amqj.logging.level
```

can be used to configure the logging level. For the broker, you can use the environment variable AMQJ_LOGGING_LEVEL which is picked up by the qpid-run script (called by qpid-server to start the broker) at runtime.

For client code that you've written, simply pass in a system property to your command line to set it to the level you'd like i.e.

```
-Damqj.logging.level=INFO
```

The log level for the broker defaults to INFO if the env variable is not set, but you may find that your log4j properties affect this. Setting the property noted above should address this.

2.1.9. How can I configure my application to use Qpid client logging?

If you don't already have a logging implementation in your classpath you should add slf4-log4j12-1.4.0.jar and log4j-1.2.12.jar.

2.1.10. How can I configure the broker ?

The broker configuration is contained in the <installed-dir>/etc/config.xml file. You can copy and edit this file and then specify your own configuration file as a parameter to the startup script using the -c flag i.e. qpid-server -c <your_config_file's_path>

For more detailed information on configuration, please see ???

2.1.11. What ports does the broker use?

The broker defaults to use port 5672 at startup for AMQP traffic. If the management interface is enabled it starts on port 8999 by default.

The JMX management interface actually requires 2 ports to operate, the second of which is indicated to the client application during connection initiation to the main (default: 8999) port. Previously this second port has been chosen at random during broker startup, however since Qpid 0.5 this has been fixed to a port 100 higher than the main port(ie Default:9099) in order to ease firewall navigation.

2.1.12. How can I change the port the broker uses at runtime ?

The broker defaults to use port 5672 at startup for AMQP traffic. The broker also uses port 8999 for the JMX Management interface.

To change the AMQP traffic port use the -p flag at startup. To change the management port use -m i.e. qpid-server -p <port_number_to_use> -m <port_number_to_use>

Use this to get round any issues on your host server with port 5672/8999 being in use/unavailable.

For additional details on what ports the broker uses see Section 2.1.11, " What ports does the broker use? " FAQ entry. For more detailed information on configuration, please see ???

2.1.13. What command line options can I pass into the qpid-server script ?

The following command line options are available:

The following options are available:

Table 7.1. Command Line Options

Option	Long Option	Description
b	bind	Bind to the specified address overriding any value in the config file
c	config	Use the given configuration file
h	help	Prints list of options

l	logconfig	Use the specified log4j.xml file rather than that in the etc directory
m	mport	Specify port to listen on for the JMX Management. Overrides value in config file
p	port	Specify port to listen on. Overrides value in config file
v	version	Print version information and exit
w	logwatch	Specify interval for checking for logging config changes. Zero means no checking

2.1.14. How do I authenticate with the broker ? What user id & password should I use ?

You should login as user guest with password guest

2.1.15. How do I create queues that will always be instantiated at broker startup ?

You can configure queues which will be created at broker startup by tailoring a copy of the virtualhosts.xml file provided in the installed qpid-version/etc directory.

So, if you're using a queue called 'devqueue' you can ensure that it is created at startup by using an entry something like this:

```
<virtualhosts>
  <default>test</default>
  <virtualhost>
   <name>test</name>
   <test>
   <queue>
      <name>devqueue</name>
      <devqueue>
        <exchange>amq.direct</exchange>
        <maximumQueueDepth>4235264</maximumQueueDepth>  <!-- 4Mb -->
        <maximumMessageSize>2117632</maximumMessageSize> <!-- 2Mb -->
        <maximumMessageAge>600000</maximumMessageAge>  <!-- 10 mins -->
      </devqueue>
   </queue>
   </test>
  </virtualhost>
</virtualhosts>
```

Note that the name (in thie example above the name is 'test') element should match the virtualhost that you're using to create connections to the broker. This is effectively a namespace used to prevent queue name clashes etc. You can also see that we've set the 'test' virtual host to be the default for any connections which do not specify a virtual host (in the <default> tag).

You can amend the config.xml to point at a different virtualhosts.xml file by editing the <virtualhosts/ > element.

So, for example, you could tell the broker to use a file in your home directory by creating a new config.xml file with the following entry:

<virtualhosts>/home/myhomedir/virtualhosts.xml</virtualhosts>

You can then pass this amended config.xml into the broker at startup using the -c flag i.e. qpid-server -c <path>/config.xml

2.1.16. How do I create queues at runtime?

Queues can be dynamically created at runtime by creating a consumer for them. After they have been created and bound (which happens automatically when a JMS Consumer is created) a publisher can send messages to them.

2.1.17. How do I tune the broker?

There are a number of tuning options available, please see the Section 8, " How to Tune M3 Java Broker Performance " page for more information.

2.1.18. Where do undeliverable messages end up ?

At present, messages with an invalid routing key will be returned to the sender. If you register an exception listener for your publisher (easiest to do by making your publisher implement the ExceptionListener interface and coding the onException method) you'll see that you end up in onException in this case. You can expect to be catching a subclass of org.apache.qpid.AMQUndeliveredException.

2.1.19. Can I configure the name of the Qpid broker log file at runtime ?

If you simply start the Qpid broker using the default configuration, then the log file is written to $QPID_WORK/log/qpid.log

This is not ideal if you want to run several instances from one install, or acrhive logs to a shared drive from several hosts.

To make life easier, there are two optional ways to configure the naming convention used for the broker log.

2.1.19.1. Setting a prefix or suffix

Users should set the following environment variables before running qpid-server:

QPID_LOG_PREFIX - will prefix the log file name with the specified value e.g. if you set this value to be the name of your host (for example) it could look something like host123qpid.log

QPID_LOG_SUFFIX - will suffix the file name with the specified value e.g. if you set this value to be the name of your application (for example) if could look something like qpidMyApp.log

2.1.19.2. Including the PID

Setting either of these variables to the special value PID will introduce the process id of the java process into the file name as a prefix or suffix as specified**

2.1.20. My client application appears to have hung?

The client code currently has various timeouts scattered throughout the code. These can cause your client to appear like it has hung when it is actually waiting for the timeout ot compelete. One example is when

the broker becomes non-responsive, the client code has a hard coded 2 minute timeout that it will wait when closing a connection. These timeouts need to be consolidated and exposed. see ???

2.1.21. How do I contact the Qpid team ?

For general questions, please subscribe to the users@qpid.apache.org [mailto:users@qpid.apache.org] mailing list.

For development questions, please subscribe to the dev@qpid.apache.org [mailto:dev@qpid.apache.org] mailing list.

More details on these lists are available on our ??? page.

2.1.22. How can I change a user's password while the broker is up ?

You can do this via the ???. To do this simply log in to the management console as an admin user (you need to have created an admin account in the jmxremote.access file first) and then select the 'UserManagement' mbean. Select the user in the table and click the Set Password button. Alternatively, update the password file and use the management console to reload the file with the button at the bottom of the 'UserManagement' view. In both cases, this will take effect when the user next logs in i.e. will not cause them to be disconnected if they are already connected.

For more information on the Management Console please see our Section 1.1.5, “ Qpid JMX Management Console User Guide ”

2.1.23. How do I know if there is a consumer for a message I am going to send?

Knowing that there is a consumer for a message is quite tricky. That said using the qpid.jms.Session#createProducer with immediate and mandatory set to true will get you part of the way there.

If you are publishing to a well known queue then immediate will let you know if there is any consumer able to pre-fetch that message at the time you send it. If not it will be returned to you on your connection listener.

If you are sending to a queue that the consumer creates then the mandatory flag will let you know if they have not yet created that queue.

These flags will not be able to tell you if the consuming application has received the message and is able to process it.

2.1.24. How do I use an InVM Broker for my own tests?

I would take a look at the testPassiveTTL in TimeToLiveTest.java [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/systests/src/main/java/org/apache/qpid/server/queue/TimeToLiveTest.java]

The setUp and tearDown methods show how to correctly start up a broker for InVM testing. If you write your tests using a file for the JNDI you can then very easily swap between running your tests InVM and against a real broker.

See our ??? on how to confgure it

Basically though you just need to set two System Properites:

java.naming.factory.initial = org.apache.qpid.jndi.PropertiesFileInitialContextFactory
java.naming.provider.url = <your JNDI file>

and call getInitialContext() in your code.

You will of course need to have the broker libraries on your class path for this to run.

2.1.25. How can I inspect the contents of my MessageStore?

There are two possibilities here:

1) The management console can be used to interogate an active broker and browse the contents of a queue.See the ??? page for further details.

2) The ??? can be used to inspect the contents of a persistent message store. Note: this can currently only be used when the broker is offline.

2.1.26. Why are my transient messages being so slow?

You should check that you aren't sending persistent messages, this is the default. If you want to send transient messages you must explicitly set this option when instantiating your MessageProducer or on the send() method.

2.1.27. Why does my producer fill up the broker with messages?

The Java broker does not currently implement producer flow control. Publishes are currently asynchronous, so there is no ability to rate limit this automatically. While this is something which will be addressed in the future, it is currently up to applications to ensure that they do not publish faster than the messages are being consumed for signifcant periods of time.

2.1.28. The broker keeps throwing an OutOfMemory exception?

The broker can no longer store any more messages in memory. This is particular evident if you are using the MemoryMessageStore. To alleviate this issue you should ensure that your clients are consuming all the messages from the broker.

You may also want to increase the memory allowance to the broker though this will only delay the exception if you are publishing messages faster than you are consuming. See ??? for details of changing the memory settings.

2.1.29. Why am I getting a broker side exception when I try to publish to a queue or a topic ?

If you get a stack trace like this when you try to publish, then you may have typo'd the exchange type in your queue or topic declaration. Open your virtualhosts.xml and check that the

```
<exchange>amq.direct</exchange>
```

```
2009-01-12 15:26:27,957 ERROR [pool-11-thread-2] protocol.AMQMinaProtocolSession (
java.lang.NullPointerException
        at org.apache.qpid.server.security.access.PrincipalPermissions.authorise(P
        at org.apache.qpid.server.security.access.plugins.SimpleXML.authorise(Simp
```

```
at org.apache.qpid.server.handler.QueueBindHandler.methodReceived(QueueBin
at org.apache.qpid.server.handler.ServerMethodDispatcherImpl.dispatchQueue
at org.apache.qpid.framing.amqp_8_0.QueueBindBodyImpl.execute(QueueBindBod
at org.apache.qpid.server.state.AMQStateManager.methodReceived(AMQStateMan
at org.apache.qpid.server.protocol.AMQMinaProtocolSession.methodFrameRecei
at org.apache.qpid.framing.AMQMethodBodyImpl.handle(AMQMethodBodyImpl.java
at org.apache.qpid.server.protocol.AMQMinaProtocolSession.frameReceived(AM
at org.apache.qpid.server.protocol.AMQMinaProtocolSession.dataBlockReceive
at org.apache.qpid.server.protocol.AMQPFastProtocolHandler.messageReceived
at org.apache.mina.common.support.AbstractIoFilterChain$TailFilter.message
at org.apache.mina.common.support.AbstractIoFilterChain.callNextMessageRec
at org.apache.mina.common.support.AbstractIoFilterChain.access$1200(Abstra
at org.apache.mina.common.support.AbstractIoFilterChain$EntryImpl$1.messag
at org.apache.qpid.pool.PoolingFilter.messageReceived(PoolingFilter.java:3
at org.apache.mina.filter.ReferenceCountingIoFilter.messageReceived(Refere
at org.apache.mina.common.support.AbstractIoFilterChain.callNextMessageRec
at org.apache.mina.common.support.AbstractIoFilterChain.access$1200(Abstra
at org.apache.mina.common.support.AbstractIoFilterChain$EntryImpl$1.messag
at org.apache.mina.filter.codec.support.SimpleProtocolDecoderOutput.flush(
at org.apache.mina.filter.codec.QpidProtocolCodecFilter.messageReceived(Qp
at org.apache.mina.common.support.AbstractIoFilterChain.callNextMessageRec
at org.apache.mina.common.support.AbstractIoFilterChain.access$1200(Abstra
at org.apache.mina.common.support.AbstractIoFilterChain$EntryImpl$1.messag
at org.apache.qpid.pool.Event$ReceivedEvent.process(Event.java:86)
at org.apache.qpid.pool.Job.processAll(Job.java:110)
at org.apache.qpid.pool.Job.run(Job.java:149)
at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecut
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.j
at java.lang.Thread.run(Thread.java:619)
```

2.1.30. Why is there a lot of AnonymousIoService threads

These threads are part of the thread pool used by Mina to process the socket. In the future we may provide tuning guidelines but at this point we have seen no performance implications from the current configuration. As the threads are part of a pool they should remain inactive until required.

2.1.31. "unable to certify the provided SSL certificate using the current SSL trust store" when connecting the Management Console to the broker.

You have not configured the console's SSL trust store properly, see ??? for more details.

2.1.32. Client keeps throwing 'Server did not respond in a timely fashion' [error code 408: Request Timeout].

Certain operations wait for a response from the Server. One such operations is commit. If the server does not respond to the commit request within a set time a Request Timeout [error code: 408] exception is thrown (Server did not respond in a timely fashion). This is to ensure that a server that has hung does not cause the client process to be come unresponsive.

However, it is possible that the server just needs a long time to process a give request. For example, sending a large persistent message when using a persistent store will take some time to a) Transfer accross the network and b) to be fully written to disk.

These situations require that the default timeout value be increased. A cilent ??? 'amqj.default_syncwrite_timeout' can be set on the client to increase the wait time. The default in 0.5 is 30000 (30s).

2.1.33. Can a use TCP_KEEPALIVE or AMQP heartbeating to keep my connection open?

See ???

3. Java Environment Variables

3.1. Setting Qpid Environment Variables

3.1.1. Qpid Deployment Path Variables

There are two main Qpid environment variables which are required to be set for Qpid deployments, QPID_HOME and QPID_WORK.

QPID_HOME - This variable is used to tell the Qpid broker where it's installed home is, which is in turn used to find dependency JARs which Qpid uses.

QPID_WORK - This variable is used by Qpid when creating all 'writeable' directories that it uses. This includes the log directory and the storage location for any BDB instances in use by your deployment (if you're using persistence with BDB). If you do not set this variable, then the broker will default (in the qpid-server script) to use the current user's homedir as the root directory for creating the writeable locations that it uses.

3.1.2. Setting Max Memory for the broker

If you simply start the Qpid broker, it will default to use a -Xmx setting of 1024M for the broker JVM. However, we would recommend that you make the maximum -Xmx heap size available, if possible, of 3Gb (for 32-bit platforms).

You can control the memory setting for your broker by setting the QPID_JAVA_MEM variable before starting the broker e.g. -Xmx3668m . Enclose your value within quotes if you also specify a -Xms value. The value in use is echo'd by the qpid-server script on startup.

4. Qpid Troubleshooting Guide

4.1. I'm getting a java.lang.UnsupportedClassVersionError when I try to start the broker. What does this mean ?

The QPID broker requires JDK 1.5 or later. If you're seeing this exception you don't have that version in your path. Set JAVA_HOME to the correct version and ensure the bin directory is on your path.

java.lang.UnsupportedClassVersionError: org/apache/qpid/server/Main (Unsupported major.minor version 49.0) at java.lang.ClassLoader.defineClass(Ljava.lang.String; [BIILjava.security.ProtectionDomain;)Ljava.lang.Class;(Unknown Source) at java.security.SecureClassLoader.defineClass(Ljava.lang.String;

[BIILjava.security.CodeSource;)Ljava.lang.Class;(SecureClassLoader.java:123) at java.net.URLClassLoader.defineClass(Ljava.lang.String;Lsun.misc.Resource;)Ljava.lang.Class; (URLClassLoader.java:251) at java.net.URLClassLoader.access $100(Ljava.net.URLClassLoader;Ljava.lang.String;Lsun.misc.Resource;)Ljava.lang.Class; (URLClassLoader.java:55) at java.net.URLClassLoader$1.run()Ljava.lang.Object; (URLClassLoader.java:194) at jrockit.vm.AccessController.do_privileged_exc(Ljava.security.PrivilegedExceptionAction;Ljava.security.AccessControlC (Unknown Source) at jrockit.vm.AccessController.doPrivileged(Ljava.security.PrivilegedExceptionAction;Ljava.security.AccessControlContext (Unknown Source) at java.net.URLClassLoader.findClass(Ljava.lang.String;)Ljava.lang.Class; (URLClassLoader.java:187) at java.lang.ClassLoader.loadClass(Ljava.lang.String;Z)Ljava.lang.Class; (Unknown Source) at sun.misc.Launcher $AppClassLoader.loadClass(Ljava.lang.String;Z)Ljava.lang.Class;(Launcher.java:274) at java.lang.ClassLoader.loadClass(Ljava.lang.String;)Ljava.lang.Class; (Unknown Source) at java.lang.ClassLoader.loadClassFromNative(II)Ljava.lang.Class; (Unknown Source)

4.2. I'm having a problem binding to the required host:port at broker startup ?

This error probably indicates that another process is using the port you the broker is trying to listen on. If you haven't amended the default configuration this will be 5672. To check what process is using the port you can use 'netstat -an |grep 5672'.

To change the port your broker uses, either edit the config.xml you are using. You can specify an alternative config.xml from the one provided in /etc by using the -c flag i.e. qpid-server -c <my config file path>.

You can also amend the port more simply using the -p option to qpid-server i.e. qpid-server -p <my port number'

4.3. I'm having problems with my classpath. How can I ensure that my classpath is ok ?

When you are running the broker the classpath is taken care of for you, via the manifest entries in the launch jars that the qpid-server configuration file adds to the classpath.

However, if you are running your own client code and experiencing classspath errors you need to ensure that the client-launch.jar from the installed Qpid lib directory is on your classpath. The manifest for this jar includes the common-launch.jar, and thus all the code you need to run a client application.

4.4. I can't get the broker to start. How can I diagnose the problem ?

Firstly have a look at the broker log file - either on stdout or in $QPID_WORK/log/qpid.log or in $HOME/log/qpid.log if you haven't set QPID_WORK.

You should see the problem logged in here via log4j and a stack trace. Have a look at the other entries on this page for common problems. If the log file includes a line like:

"2006-10-13 09:58:14,672 INFO [main] server.Main (Main.java:343) - Qpid.AMQP listening on non-SSL address 0.0.0.0/0.0.0.0:5672"

... then you know the broker started up. If not, then it didn't.

4.5. When I try to send messages to a queue I'm getting a error as the queue does not exist. What can I do ?

In Qpid queues need a consumer before they really exist, unless you have used the virtualhosts.xml file to specify queues which should always be created at broker startup. If you don't want to use this config, then simply ensure that you consume first from queue before staring to publish to it. See the entry on our ??? for more details of using the virtualhosts.xml route.

Chapter 8. How Tos

1. Add New Users

The Qpid Java Broker has a single reference source (???) that defines all the users in the system.

To add a new user to the broker the password file must be updated. The details about adding entries and when these updates take effect are dependent on the file format each of which are described below.

1.1. Available Password file formats

There are currently two different file formats available for use depending on the PrincipalDatabase that is desired. In all cases the clients need not be aware of the type of PrincipalDatabase in use they only need support the SASL mechanisms they provide.

- Section 1.1.1, “ Plain ”
- Section 1.1.3, “ Base64MD5 Password File Format ”

1.1.1. Plain

The plain file has the following format:

```
# Plain password authentication file.
# default name : passwd
# Format <username>:<password>
#e.g.
martin:password
```

As the contents of the file are plain text and the password is taken to be everything to the right of the ':'(colon). The password, therefore, cannot contain a ':' colon, but this can be used to delimit the password.

Lines starting with a '#' are treated as comments.

1.1.2. Where is the password file for my broker ?

The location of the password file in use for your broker is as configured in your config.xml file.

```
<principal-databases>
            <principal-database>
                <name>passwordfile</name>
                <class>org.apache.qpid.server.security.auth.database.PlainPassword
                <attributes>
                    <attribute>
                        <name>passwordFile</name>
                        <value>${conf}/passwd</value>
                    </attribute>
                </attributes>
            </principal-database>
        </principal-databases>
```

So in the example config.xml file this password file lives in the directory specified as the conf directory (at the top of your config.xml file).

If you wish to use Base64 encoding for your password file, then in the <class> element above you should specify org.apache.qpid.server.security.auth.database.Base64MD5PasswordFilePrincipalDatabase

The default is:

```
<conf>${prefix}/etc</conf>
```

1.1.3. Base64MD5 Password File Format

This format can be used to ensure that SAs cannot read the plain text password values from your password file on disk.

The Base64MD5 file uses the following format:

```
# Base64MD5 password authentication file
# default name : qpid.passwd
# Format <username>:<Base64 Encoded MD5 hash of the users password>
#e.g.
martin:X03MO1qnZdYdgyfeuILPmQ==
```

As with the Plain format the line is delimited by a ':'(colon). The password field contains the MD5 Hash of the users password encoded in Base64.

This file is read on broker start-up and is not re-read.

1.1.4. How can I update a Base64MD5 password file ?

To update the file there are two options:

1. Edit the file by hand using the *qpid-passwd* tool that will generate the required lines. The output from the tool is the text that needs to be copied in to your active password file. This tool is located in the broker bin directory. Eventually it is planned for this tool to emulate the functionality of ??? for qpid passwd files. *NOTE:* For the changes to be seen by the broker you must either restart the broker or reload the data with the management tools (see Section 1.1.5, “ Qpid JMX Management Console User Guide ”)

2. Use the management tools to create a new user. The changes will be made by the broker to the password file and the new user will be immediately available to the system (see Section 1.1.5, “ Qpid JMX Management Console User Guide ”).

1.2. Dynamic changes to password files.

The Plain password file and the Base64MD5 format file are both only read once on start up.

To make changes dynamically there are two options, both require administrator access via the Management Console (see Section 1.1.5, “ Qpid JMX Management Console User Guide ”)

1. You can replace the file and use the console to reload its contents.

2. The management console provides an interface to create, delete and amend the users. These changes are written back to the active password file.

1.3. How password files and PrincipalDatabases relate to authentication mechanisms

For each type of password file a PrincipalDatabase exists that parses the contents. These PrincipalDatabases load various SASL mechanism based on their supportability. e.g. the Base64MD5 file format can't support Plain authentication as the plain password is not available. Any client connecting need only be concerned about the SASL module they support and not the type of PrincipalDatabase. So I client that understands CRAM-MD5 will work correctly with a Plain and Base64MD5 PrincipalDatabase.

Table 8.1. File Format and Principal Database

FileFormat/PrincipalDatabase	SASL
Plain	AMQPLAIN PLAIN CRAM-MD5
Base64MD5	CRAM-MD5 CRAM-MD5-HASHED

For details of SASL support see ???

2. Configure ACLs

2.1. Configure ACLs

2.1.1. Specification

- ???
- Section 9, “ ACL ”

2.1.2. C++ Broker

The C++ broker supports Section 9, “ ACL ” of the ACLs

2.1.3. Java Broker

- ???
- Support for Version 2 specification is in progress.

3. Configure Java Qpid to use a SSL connection.

3.1. Using SSL connection with Qpid Java.

This section will show how to use SSL to enable secure connections between a Java client and broker.

3.2. Setup

3.2.1. Broker Setup

The broker configuration file (config.xml) needs to be updated to include the SSL keystore location details.

```
<!-- Additions required to Connector Section -->

<ssl>
    <enabled>true</enabled>
    <sslOnly>true</sslOnly>
    <keystorePath>/path/to/keystore.ks</keystorePath>
    <keystorePassword>keystorepass</keystorePassword>
</ssl>
```

The sslOnly option is included here for completeness however this will disable the unencrypted port and leave only the SSL port listening for connections.

3.2.2. Client Setup

The best place to start looking is class *SSLConfiguration* this is provided to the connection during creation however there is currently no example that demonstrates its use.

3.3. Performing the connection.

4. Configure Log4j CompositeRolling Appender

4.1. How to configure the CompositeRolling log4j Appender

There are several sections of our default log4j file that will need your attention if you wish to fully use this Appender.

1. Enable the Appender

   The default log4j.xml file uses the FileAppender, swap this for the ArchivingFileAppender as follows:

   ```
   <!-- Log all info events to file -->
   <root>
       <priority value="info"/>

       <appender-ref ref="ArchivingFileAppender"/>
   </root>
   ```

2. Configure the Appender

   The Appender has a number of parameters that can be adjusted depending on what you are trying to achieve. For clarity lets take a quick look at the complete default appender:

   ```
   <appender name="ArchivingFileAppender" class="org.apache.log4j.QpidCompositeRo
         <!-- Ensure that logs allways have the dateFormat set-->
         <param name="StaticLogFileName" value="false"/>
         <param name="File" value="${QPID_WORK}/log/${logprefix}qpid${logsuffix}.
         <param name="Append" value="false"/>
         <!-- Change the direction so newer files have bigger numbers -->
   ```

```
        <!-- So log.1 is written then log.2 etc This prevents a lot of file rena
        <param name="CountDirection" value="1"/>
        <!-- Use default 10MB -->
        <!--param name="MaxFileSize" value="100000"/-->
        <param name="DatePattern" value="'.'yyyy-MM-dd-HH-mm"/>
        <!-- Unlimited number of backups -->
        <param name="MaxSizeRollBackups" value="-1"/>
        <!-- Compress(gzip) the backup files-->
        <param name="CompressBackupFiles" value="true"/>
        <!-- Compress the backup files using a second thread -->
        <param name="CompressAsync" value="true"/>
        <!-- Start at zero numbered files-->
        <param name="ZeroBased" value="true"/>
        <!-- Backup Location -->
        <param name="backupFilesToPath" value="${QPID_WORK}/backup/log"/>

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) -
        </layout>
    </appender>
```

The appender configuration has three groups of parameter configuration.

The first group is for configuration of the file name. The default is to write a log file to QPID_WORK/log/qpid.log (Remembering you can use the logprefix and logsuffix values to modify the file name, see Property Config).

```
        <!-- Ensure that logs always have the dateFormat set-->
        <param name="StaticLogFileName" value="false"/>
        <param name="File" value="${QPID_WORK}/log/${logprefix}qpid${logsuffix}.
        <param name="Append" value="false"/>
```

The second section allows the specification of a Maximum File Size and a DatePattern that will be used to move on to the next file.

When MaxFileSize is reached a new log file will be created The DataPattern is used to decide when to create a new log file, so here a new file will be created for every minute and every 10Meg of data. So if 15MB of data is made every minute then there will be two log files created each minute. One at the start of the minute and a second when the file hit 10MB. When the next minute arrives a new file will be made even though it only has 5MB of content. For a production system it would be expected to be changed to something like 'yyyy-MM-dd' which would make a new log file each day and keep the files to a max of 10MB.

The final MaxSizeRollBackups allows you to limit the amount of disk you are using by only keeping the last n backups.

```
        <!-- Change the direction so newer files have bigger numbers -->
        <!-- So log.1 is written then log.2 etc This prevents a lot of file rena
        <param name="CountDirection" value="1"/>
        <!-- Use default 10MB -->
        <!--param name="MaxFileSize" value="100000"/-->
        <param name="DatePattern" value="'.'yyyy-MM-dd-HH-mm"/>
        <!-- Unlimited number of backups -->
```

```
<param name="MaxSizeRollBackups" value="-1"/>
```

The final section allows the old log files to be compressed and copied to a new location.

```
<!-- Compress(gzip) the backup files-->
<param name="CompressBackupFiles" value="true"/>
<!-- Compress the backup files using a second thread -->
<param name="CompressAsync" value="true"/>
<!-- Start at zero numbered files-->
<param name="ZeroBased" value="true"/>
<!-- Backup Location -->
<param name="backupFilesToPath" value="${QPID_WORK}/backup/log"/>
```

5. Configure the Broker via config.xml

5.1. Broker config.xml Overview

The broker config.xml file which is shipped in the etc directory of any Qpid binary distribution details various options and configuration for the Java Qpid broker implementation.

In tandem with the virtualhosts.xml file, the config.xml file allows you to control much of the deployment detail for your Qpid broker in a flexible fashion.

Note that you can pass the config.xml you wish to use for your broker instance to the broker using the -c command line option. In turn, you can specify the paths for the broker password file and virtualhosts.xml files in your config.xml for simplicity.

For more information about command line configuration options please see ???.

5.2. Qpid Version

The config format has changed between versions here you can find the configuration details on a per version basis.

??? ???

6. Configure the Virtual Hosts via virtualhosts.xml

6.1. virtualhosts.xml Overview

This configuration file contains details of all queues and topics, and associated properties, to be created on broker startup. These details are configured on a per virtual host basis.

Note that if you do not add details of a queue or topic you intend to use to this file, you must first create a consumer on a queue/topic before you can publish to it using Qpid.

Thus most application deployments need a virtualhosts.xml file with at least some minimal detail.

6.1.1. XML Format with Comments

The virtualhosts.xml which currently ships as part of the Qpid distribution is really targeted at development use, and supports various artifacts commonly used by the Qpid development team.

As a result, it is reasonably complex. In the example XML below, I have tried to simplify one example virtual host setup which is possibly more useful for new users of Qpid or development teams looking to simply make use of the Qpid broker in their deployment.

I have also added some inline comments on each section, which should give some extra information on the purpose of the various elements.

```
<virtualhosts>
    <!-- Sets the default virtual host for connections which do not specify a vh -
    <default>localhost</default>
    <!-- Define a virtual host and all it's config -->
    <virtualhost>
        <name>localhost</name>
        <localhost>
            <!-- Define the types of additional AMQP exchange available for this v
            <!-- Always get amq.direct (for queues) and amq.topic (for topics) by
            <exchanges>
                <!-- Example of declaring an additional exchanges type for develop
                <exchange>
                    <type>direct</type>
                    <name>test.direct</name>
                    <durable>true</durable>
                </exchange>
            </exchanges>

            <!-- Define the set of queues to be created at broker startup -->
            <queues>
                <!-- The properties configured here will be applied as defaults to
                <!-- queues subsequently defined unless explicitly overridden -->
                <exchange>amq.direct</exchange>
                <!-- Set threshold values for queue monitor alerting to log -->
                <maximumQueueDepth>4235264</maximumQueueDepth>  <!-- 4Mb -->
                <maximumMessageSize>2117632</maximumMessageSize> <!-- 2Mb -->
                <maximumMessageAge>600000</maximumMessageAge>  <!-- 10 mins -->

                <!-- Define a queue with all default settings -->
                <queue>
                    <name>ping</name>
                </queue>
                <!-- Example definitions of queues with overriden settings -->
                <queue>
                    <name>test-queue</name>
                    <test-queue>
                        <exchange>test.direct</exchange>
                        <durable>true</durable>
                    </test-queue>
                </queue>
                <queue>
```

```
                    <name>test-ping</name>
                    <test-ping>
                        <exchange>test.direct</exchange>
                    </test-ping>
                </queue>
            </queues>
        </localhost>
    </virtualhost>
</virtualhosts>
```

6.1.2. Using your own virtualhosts.xml

Note that the config.xml file shipped as an example (or developer default) in the Qpid distribution contains an element which defines the path to the virtualhosts.xml.

When using your own virtualhosts.xml you must edit this path to point at the location of your file.

7. Debug using log4j

7.1. Debugging with log4j configurations

Unfortunately setting of logging in the Java Broker is not simply a matter of setting one of WARN,INFO,DEBUG. At some point in the future we may have more BAU logging that falls in to that category but more likely is that we will have a varioius config files that can be swapped in (dynamically) to understand what is going on.

This page will be host to a variety of useful configuration setups that will allow a user or developer to extract only the information they are interested in logging. Each section will be targeted at logging in a particular area and will include a full log4j file that can be used. In addition the logging *category* elements will be presented and discussed so that the user can create their own file.

Currently the configuration that is available has not been fully documented and as such there are gaps in what is desired and what is available. Some times this is due to the desire to reduce the overhead in message processing, but sometimes it is simply an oversight. Hopefully in future releases the latter will be addressed but care needs to be taken when adding logging to the 'Message Flow' path as this will have performance implications.

7.1.1. Logging Connection State *Deprecated*

deprecation notice Version 0.6 of the Java broker includes ??? functionality which improves upon these messages and as such enabling status logging would be more beneficial. The configuration file has been left here for assistence with broker versions prior to 0.6.

The goals of this configuration are to record:

- New Connections
- New Consumers
- Identify slow consumers
- Closing of Consumers
- Closing of Connections

An additional goal of this configuration is to minimise any impact to the 'message flow' path. So it should not adversely affect production systems.

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="FileAppender" class="org.apache.log4j.FileAppender">
        <param name="File" value="${QPID_WORK}/log/${logprefix}qpid${logsuffix}.lo
        <param name="Append" value="false"/>

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m
        </layout>

    </appender>

    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m
        </layout>
    </appender>

    <category name="Qpid.Broker">

        <priority value="debug"/>
    </category>


    <!-- Provide warnings to standard output -->
    <category name="org.apache.qpid">
        <priority value="warn"/>
    </category>


    <!-- Connection Logging -->

    <!-- Log details of client starting connection -->
    <category name="org.apache.qpid.server.handler.ConnectionStartOkMethodHandler"
        <priority value="info"/>
    </category>
    <!-- Log details of client closing connection -->
    <category name="org.apache.qpid.server.handler.ConnectionCloseMethodHandler">
        <priority value="info"/>
    </category>
    <!-- Log details of client responding to be asked to closing connection -->

    <category name="org.apache.qpid.server.handler.ConnectionCloseOkMethodHandler"
        <priority value="info"/>
    </category>


    <!-- Consumer Logging -->
    <!-- Provide details of Consumers connecting-->
```

```
    <category name="org.apache.qpid.server.handler.BasicConsumeMethodHandler">
        <priority value="debug"/>
    </category>

    <!-- Provide details of Consumers disconnecting, if the call it-->
    <category name="org.apache.qpid.server.handler.BasicCancelMethodHandler">
        <priority value="debug"/>
    </category>
    <!-- Provide details of when a channel closes to attempt to match to the Consu
    <category name="org.apache.qpid.server.handler.ChannelCloseHandler">
        <priority value="info"/>
    </category>

    <!-- Provide details of Consumers starting to consume-->
    <category name="org.apache.qpid.server.handler.ChannelFlowHandler">
        <priority value="debug"/>
    </category>
    <!-- Provide details of what consumers are going to be consuming-->
    <category name="org.apache.qpid.server.handler.QueueBindHandler">
        <priority value="info"/>
    </category>

    <!-- No way of determining if publish message is returned, client log should s

    <root>
        <priority value="debug"/>
        <appender-ref ref="STDOUT"/>
        <appender-ref ref="FileAppender"/>
    </root>

</log4j:configuration>
```

7.1.2. Debugging My Application

This is the most often asked for set of configuration. The goals of this configuration are to record:

- New Connections
- New Consumers
- Message Publications
- Message Consumption
- Identify slow consumers
- Closing of Consumers
- Closing of Connections

NOTE: This configuration enables message logging on the 'message flow' path so should only be used were message volume is low. *Every message that is sent to the broker will generate at least four logging statements*

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="FileAppender" class="org.apache.log4j.FileAppender">
        <param name="File" value="${QPID_WORK}/log/${logprefix}qpid${logsuffix}.lo
        <param name="Append" value="false"/>

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m
        </layout>

    </appender>

    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m
        </layout>
    </appender>

    <category name="Qpid.Broker">

        <priority value="debug"/>
    </category>


    <!-- Provide warnings to standard output -->
    <category name="org.apache.qpid">
        <priority value="warn"/>
    </category>


    <!-- Connection Logging -->

    <!-- Log details of client starting connection -->
    <category name="org.apache.qpid.server.handler.ConnectionStartOkMethodHandler"
        <priority value="info"/>
    </category>
    <!-- Log details of client closing connection -->
    <category name="org.apache.qpid.server.handler.ConnectionCloseMethodHandler">
        <priority value="info"/>
    </category>
    <!-- Log details of client responding to be asked to closing connection -->

    <category name="org.apache.qpid.server.handler.ConnectionCloseOkMethodHandler"
        <priority value="info"/>
    </category>

    <!-- Consumer Logging -->
    <!-- Provide details of Consumers connecting-->
    <category name="org.apache.qpid.server.handler.BasicConsumeMethodHandler">
        <priority value="debug"/>
    </category>

    <!-- Provide details of Consumers disconnecting, if the call it-->
```

```
    <category name="org.apache.qpid.server.handler.BasicCancelMethodHandler">
        <priority value="debug"/>
    </category>
    <!-- Provide details of when a channel closes to attempt to match to the Consu
    <category name="org.apache.qpid.server.handler.ChannelCloseHandler">
        <priority value="info"/>
    </category>

    <!-- Provide details of Consumers starting to consume-->
    <category name="org.apache.qpid.server.handler.ChannelFlowHandler">
        <priority value="debug"/>
    </category>
    <!-- Provide details of what consumers are going to be consuming-->
    <category name="org.apache.qpid.server.handler.QueueBindHandler">
        <priority value="info"/>
    </category>

    <!-- No way of determining if publish message is returned, client log should s

    <!-- WARNING DO NOT ENABLE THIS IN PRODUCTION -->
    <!-- Will generate minimum one log statements per published message -->
    <!-- Will generate will log receiving of all body frame, count will vary on si
    <!-- Empty Message = no body, Body is up to 64kb of data -->
    <!-- Will generate three log statements per recevied message -->

    <!-- Log messages flow-->
    <category name="org.apache.qpid.server.AMQChannel">

        <priority value="debug"/>
    </category>

    <root>
        <priority value="debug"/>
        <appender-ref ref="STDOUT"/>
        <appender-ref ref="FileAppender"/>
    </root>

</log4j:configuration>
```

8. How to Tune M3 Java Broker Performance

8.1. Problem Statement

During destructive testing of the Qpid M3 Java Broker, we tested some tuning techniques and deployment changes to improve the Qpid M3 Java Broker's capacity to maintain high levels of throughput, particularly in the case of a slower consumer than produceer (i.e. a growing backlog).

The focus of this page is to detail the results of tuning & deployment changes trialled.

The successful tuning changes are applicable for any deployment expecting to see bursts of high volume throughput (1000s of persistent messages in large batches). Any user wishing to use these options *must test them thoroughly in their own environment with representative volumes*.

8.2. Successful Tuning Options

The key scenario being taregetted by these changes is a broker under heavy load (processing a large batch of persistent messages)can be seen to perform slowly when filling up with an influx of high volume transient messages which are queued behind the persistent backlog. However, the changes suggested will be equally applicable to general heavy load scenarios.

The easiest way to address this is to separate streams of messages. Thus allowing the separate streams of messages to be processed, and preventing a backlog behind a particular slow consumer.

These strategies have been successfully tested to mitigate this problem:

Table 8.2.

Strategy	Result
Seperate connections to one broker for separate streams of messages.	Messages processed successfully, no problems experienced
Seperate brokers for transient and persistent messages.	Messages processed successfully, no problems experienced

Separate Connections Using separate connections effectively means that the two streams of data are not being processed via the same buffer, and thus the broker gets & processes the transient messages while processing the persistent messages. Thus any build up of unprocessed data is minimal and transitory.

Separate Brokers Using separate brokers may mean more work in terms of client connection details being changed, and from an operational perspective. However, it is certainly the most clear cut way of isolating the two streams of messages and the heaps impacted.

8.2.1. Additional tuning

It is worth testing if changing the size of the Qpid read/write thread pool improves performance (eg. by setting JAVA_OPTS="-Damqj.read_write_pool_size=32" before running qpid-server). By default this is equal to the number of CPU cores, but a higher number may show better performance with some work loads.

It is also important to note that you should give the Qpid broker plenty of memory - for any serious application at least a -Xmx of 3Gb. If you are deploying on a 64 bit platform, a larger heap is definitely worth testing with. We will be testing tuning options around a larger heap shortly.

8.3. Next Steps

These two options have been testing using a Qpid test case, and demonstrated that for a test case with a profile of persistent heavy load following by constant transient high load traffic they provide significant improvment.

However, the deploying project *must* complete their own testing, using the same destructive test cases, representative message paradigms & volumes, in order to verify the proposed mitigation options.

The using programme should then choose the option most applicable for their deployment and perform BAU testing before any implementation into a production or pilot environment.

9. Qpid Java Build How To

9.1. Build Instructions - General

9.1.1. Check out the source

Firstly, check the source for Qpid out of our subversion repository:

???

9.1.2. Prerequisites

For the broker code you need JDK 1.5.0_15 or later. You should set JAVA_HOME and include the bin directory in your PATH.

Check it's ok by executing java -v !

If you are wanting to run the python tests against the broker you will of course need a version of python.

9.2. Build Instructions - Trunk

Our build system has reverted to ant as of May 2008.

The ant target 'help' will tell you what you need to know about the build system.

9.2.1. Ant Build Scripts

Currently the Qpid java project builds using ant.

The ant build system is set up in a modular way, with a top level build script and template for module builds and then a module level build script which inherits from the template.

So, at the top level there are:

Table 8.3.

File	Description
build.xml	Top level build file for the project which defines all the build targets
common.xml	Common properties used throughout the build system
module.xml	Template used by all modules which sets up properties for module builds

Then, in each module subdirectory there is:

Table 8.4.

File	Description
build.xml	Defines all the module values for template properties

9.2.2. Build targets

The main build targets you are probably interested in are:

Table 8.5.

Target	Description
build	Builds all source code for Qpid
test	Runs the testsuite for Qpid

So, if you just want to compile everything you should run the build target in the top level build.xml file.

If you want to build an installable version of Qpid, run the archive task from the top level build.xml file.

If you want to compile an individual module, simply run the build target from the appropriate module e.g. to compile the broker source

9.2.3. Configuring Eclipse

1. Run the ant build from the root directory of Java trunk. 2. New project -> create from existing file system for broker, common, client, junit-toolkit, perftests, systests and each directory under management 4. Add the contents of lib/ to the build path 5. Setup Generated Code 6. Setup Dependencies

9.2.3.1. Generated Code

The Broker and Common packages both depend on generated code. After running 'ant' the build/scratch directory will contain this generated code. For the broker module add build/scratch/broker/src For the common module add build/scratch/common/src

9.2.3.2. Dependencies

These dependencies are correct at the time of writting however, if things are not working you can check the dependencies by looking in the modules build.xml file:

```
for i in `find . -name build.xml` ; do echo "$i:"; grep module.depends $i ; done
```

The *module.depend* value will detail which other modules are dependencies.

broker

- common
- management/common

client

- Common

systest

- client
- management/common

- broker
- broker/test
- common
- junit-toolkit
- management/tools/qpid-cli

perftests

- systests
- client
- broker
- common
- junit-toolkit

management/eclipse-plugin

- broker
- common
- management/common

management/console

- common
- client

management/agent

- common
- client

management/tools/qpid-cli

- common
- management/common

management/client

- common
- client

integrationtests

- systests
- client

- common
- junit-toolkit

testkit

- client
- broker
- common

tools

- client
- common

client/examples

- common
- client

broker-plugins

- client
- management/common
- broker
- common
- junit-toolkit

9.2.4. What next ?

If you want to run your built Qpid package, see our ??? for details of how to do that.

If you want to run our tests, you can use the ant test or testreport (produces a useful report) targets.

10. Use Priority Queues

10.1. General Information

The Qpid M3 release introduces priority queues into the Java Messaging Broker, supporting JMS clients who wish to make use of priorities in their messaging implementation.

There are some key points around the use of priority queues in Qpid, discussed in the sections below.

10.2. Defining Priority Queues

You must define a priority queue specifically before you start to use it. You cannot subsequently change a queue to/from a priority queue (without deleting it and re-creating).

You define a queue as a priority queue in the virtualhost configuration file, which the broker loads at startup. When defining the queue, add a <priority>true</priority> element. This will ensure that the queue has 10 distinct priorities, which is the number supported by JMS.

If you require fewer priorities, it is possible to specify a <priorities>int</priorities> element (where int is a valid integer value between 2 and 10 inclusive) which will give the queue that number of distinct priorities. When messages are sent to that queue, their effective priority will be calculated by partitioning the priority space. If the number of effective priorities is 2, then messages with priority 0-4 are treated the same as "lower priority" and messages with priority 5-9 are treated equivalently as "higher priority".

```
<queue>
    <name>test</name>
    <test>
        <exchange>amq.direct</exchange>
        <priority>true</priority>
    </test>
</queue>
```

10.3. Client configuration/messaging model for priority queues

There are some other configuration & paradigm changes which are required in order that priority queues work as expected.

10.3.1. Set low pre-fetch

Qpid clients receive buffered messages in batches, sized according to the pre-fetch value. The current default is 5000.

However, if you use the default value you will probably *not* see desirable behaviour with messages of different priority. This is because a message arriving after the pre-fetch buffer has filled will not leap frog messages of lower priority. It will be delivered at the front of the next batch of buffered messages (if that is appropriate), but this is most likely NOT what you need.

So, you need to set the prefetch values for your client (consumer) to make this sensible. To do this set the java system property max_prefetch on the client environment (using -D) before creating your consumer.

Setting the Qpid pre-fetch to 1 for your client means that message priority will be honoured by the Qpid broker as it dispatches messages to your client. A default for all client connections can be set via a system property:

```
-Dmax_prefetch=1
```

The prefetch can be also be adjusted on a per connection basis by adding a 'maxprefetch' value to the Section 1.2, “ Connection URL Format ”

```
amqp://guest:guest@client1/development?maxprefetch='1'&brokerlist='tcp://localhost
```

There is a slight performance cost here if using the receive() method and you could test with a slightly higher pre-fetch (up to 10) if the trade-off between throughput and prioritisation is weighted towards the former for your application. (If you're using OnMessage() then this is not a concern.)

10.3.2. Single consumer per session

If you are using the receive() method to consume messages then you should also only use one consumer per session with priority queues. If you're using OnMessage() then this is not a concern.

Chapter 9. Qpid JMX Management Console

1. Qpid JMX Management Console

1.1. Overview

The Qpid JMX Management Console is a standalone Eclipse RCP application that communicates with the broker using JMX.

1.1.1. Configuring Management Users

The Qpid Java broker has a single source of users for the system. So a user can connect to the broker to send messages and via the JMX console to check the state of the broker.

1.1.1.1. Adding a new management user

The broker does have some minimal configuration available to limit which users can connect to the JMX console and what they can do when they are there.

There are two steps required to add a new user with rights for the JMX console.

1. Create a new user login, see HowTo:???
2. Grant the new user permission to the JMX Console

1.1.1.1.1. Granting JMX Console Permissions

By default new users do not have access to the JMX console. The access to the console is controlled via the file *jmxremote.access*.

This file contains a mapping from user to privilege.

There are three privileges available:

1. readonly - The user is able to log in and view queues but not make any changes.
2. readwrite - Grants user ability to read and write queue attributes such as alerting values.
3. admin - Grants the user full access including ability to edit Users and JMX Permissions in addition to readwrite access.

This file is read at start up and can forcibly be reloaded by an admin user through the management console.

1.1.1.1.2. Access File Format

The file is a standard Java properties file and has the following format

```
<username>=<privilege>
```

If the username value is not a valid user (list in the specified PrincipalDatabase) then the broker will print a warning when it reads the file as that entry will have no meaning.

Only when the the username exists in both the access file and the PrincipalDatabase password file will the user be able to login via the JMX Console.

1.1.1.1.2.1. Example File

The file will be timestamped by the management console if edited through the console.

```
#Generated by JMX Console : Last edited by user:admin
#Tue Jun 12 16:46:39 BST 2007
admin=admin
guest=readonly
user=readwrite
```

1.1.2. Configuring Qpid JMX Management Console

1.1.2.1. Configuring Qpid JMX Management Console

Qpid has a JMX management interface that exposes a number of components of the running broker. You can find out more about the features exposed by the JMX interfaces ???.

1.1.2.1.1. Installing the Qpid JMX Management Console

1. Unzip the archive to a suitable location.

SSL encrypted connections

Recent versions of the broker can make use of SSL to encrypt their RMI based JMX connections. If a broker being connected to is making use of this ability then additional console configuration may be required, particularly when using self-signed certificates. See ??? for details.

JMXMP based connections

In previous releases of Qpid (M4 and below) the broker JMX connections could make use of the JMXMPConnector for additional security over its default RMI based JMX configuration. This is no longer the case, with SSL encrypted RMI being the favored approach going forward. However, if you wish to connect to an older broker using JMXMP the console will support this so long as the *jmxremote_optional.jar* file is provided to it. For details see ???.

1.1.2.1.2. Running the Qpid JMX Management Console

The console can be started in the following way, depending on platform:

- Windows: by running the 'qpidmc.exe' executable file.
- Linux: by running the 'qpidmc' executable.
- Mac OS X: by launching the consoles application bundle (.app file).

1.1.2.1.3. Using the Qpid JMX Management Console

Please see Section 1.1.5, “ Qpid JMX Management Console User Guide ” for details on using this Eclipse RCP application.

1.1.2.2. Using JConsole

See ???

1.1.2.3. Using HermesJMS

HermesJMS also offers integration with the Qpid management interfaces. You can get instructions and more information from HermesJMS [http://cwiki.apache.org/confluence/display/qpid/HermesJMS].

1.1.2.4. Using MC4J

MC4J [qpid_www.mc4j.org] is an alternative management tool. It provide a richer "dashboard" that can customise the raw MBeans.

1.1.2.4.1. Installation

- First download and install MC4J for your platform. Version 1.2 beta 9 is the latest version that has been tested.
- Copy the directory blaze/java/management/mc4j into the directory <MC4J-Installation>/dashboards

1.1.2.4.2. Configuration

You should create a connection the JVM to be managed. Using the Management->Create Server Connection menu option. The connection URL should be of the form: service:jmx:rmi:///jndi/rmi://localhost:8999/jmxrmi making the appropriate host and post changes.

1.1.2.4.3. Operation

You can view tabular summaries of the queues, exchanges and connections using the Global Dashboards->QPID tree view. To drill down on individual beans you can right click on the bean. This will show any available graphs too.

1.1.3. Management Console Security

1.1.3.1. Management Console Security

- Section 1.1.3.1.1, “ SSL encrypted RMI (0.5 and above) ”
- Section 1.1.3.1.2, “ JMXMP (M4 and previous) ”
- Section 1.1.3.1.3, “ User Accounts & Access Rights ”

1.1.3.1.1. SSL encrypted RMI (0.5 and above)

Current versions of the broker make use of SSL encryption to secure their RMI based JMX ConnectorServer for security purposes. This ships enabled by default, although the test SSL keystore used during development is not provided for security reasons (using this would provide no security as anyone could have access to it).

1.1.3.1.1.1. Broker Configuration

The broker configuration must be updated before the broker will start. This can be done either by disabling the SSL support, utilizing a purchased SSL certificate to create a keystore of your own, or using the example 'create-example-ssl-stores' script in the brokers bin/ directory to generate a self-signed keystore.

The broker must be configured with a keystore containing the private and public keys associated with its SSL certificate. This is accomplished by setting the Java environment properties *javax.net.ssl.keyStore* and *javax.net.ssl.keyStorePassword* respectively with the location and password of an appropriate SSL keystore. Entries for these properties exist in the brokers main configuration file alongside the other management settings (see below), although the command line options will still work and take precedence over the configuration file.

```
<management>
    <ssl>
        <enabled>true</enabled>
        <!-- Update below path to your keystore location, eg ${conf}/qpid.keystore
        <keyStorePath>${prefix}/../test_resources/ssl/keystore.jks</keyStorePath>
        <keyStorePassword>password</keyStorePassword>
    </ssl>
</management>
```

1.1.3.1.1.2. JMX Management Console Configuration

If the broker makes use of an SSL certificate signed by a known signing CA (Certification Authority), the management console needs no extra configuration, and will make use of Java's built-in CA truststore for certificate verification (you may however have to update the system-wide default truststore if your CA is not already present in it).

If however you wish to use a self-signed SSL certificate, then the management console must be provided with an SSL truststore containing a record for the SSL certificate so that it is able to validate it when presented by the broker. This is performed by setting the *javax.net.ssl.trustStore* and *javax.net.ssl.trustStorePassword* environment variables when starting the console. This can be done at the command line, or alternatively an example configuration has been made within the console's qpidmc.ini launcher configuration file that may pre-configured in advance for repeated usage. See the Section 1.1.5, " Qpid JMX Management Console User Guide " for more information on this configuration process.

1.1.3.1.1.3. JConsole Configuration

As with the JMX Management Console above, if the broker is using a self-signed SSL certificate then in order to connect remotely using JConsole, an appropriate trust store must be provided at startup. See ??? for further details on configuration.

1.1.3.1.1.4. Additional Information

More information on Java's handling of SSL certificate verification and customizing the keystores can be found in the http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CustomizingStores.

1.1.3.1.2. JMXMP (M4 and previous)

In previous releases of Qpid (M4 and below) the broker, can make use of Sun's Java Management Extensions Messaging Protocol (JMXMP) to provide encryption of the JMX connection, offering increased security over the default unencrypted RMI based JMX connection.

1.1.3.1.2.1. Download and Install

This is possible by adding the jmxremote_optional.jar as provided by Sun. This jar is covered by the Sun Binary Code License and is not compatible with the Apache License which is why this component is not bundled with Qpid.

Download the JMX Remote API 1.0.1_04 Reference Implementation from ???. The included 'jmxremote-1_0_1-bin\lib\jmxremote_optional.jar' file must be added to the broker classpath:

First set your classpath to something like this:

```
CLASSPATH=jmxremote_optional.jar
```

Then, run qpid-server passing the following additional flag:

```
qpid-server -run:external-classpath=first
```

Following this the configuration option can be updated to enabled use of the JMXMP based JMXConnectorServer.

1.1.3.1.2.2. Broker Configuration

To enabled this security option change the *security-enabled* value in your broker configuration file.

```
<management>
    <security-enabled>true</security-enabled>
</management>
```

You may also (for M2 and earlier) need to set the following system properties using the environment variable QPID_OPTS:

QPID_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8999 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"

1.1.3.1.2.3. JMX Management Console Configuration

If you wish to connect to a broker configured to use JMXMP then the console also requires provision of the Optional sections of the JMX Remote API that are not included within the JavaSE platform.

In order to make it available to the console, place the 'jmxremote_optional.jar' (rename the file if any additional information is present in the file name) jar file within the 'plugins/jmxremote.sasl_1.0.1/' folder of the console release (on Mac OS X you will need to select 'Show package contents' from the context menu whilst selecting the management console bundle in order to reveal the inner file tree).

Following the the console will automatically load the JMX Remote Optional classes and attempt the JMXMP connection when connecting to a JMXMP enabled broker.

1.1.3.1.3. User Accounts & Access Rights

In order to access the management operations via JMX, users must have an account and have been assigned appropriate access rights. See ???

1.1.4. Qpid JMX Management Console FAQ

1.1.4.1. Errors

1.1.4.1.1. How do I connect the management console to my broker using security ?

The ??? page will give you the instructions that you should use to set this up.

1.1.4.1.2. I am unable to connect Qpid JMX MC/JConsole to a remote broker running on Linux, but connecting to localhost on that machine works ?

The RMI based JMX ConnectorServer used by the broker requries two ports to operate. The console connects to an RMI Registry running on the primary (default 8999) port and retrieves the information actually needed to connect to the JMX Server. This information embeds the hostname of the remote machine, and if this is incorrect or unreachable by the connecting client the connection will fail.

This situation arises due to the hostname configuration on Linux and is generally encountered when the remote machine does not have a DNS hostname entry on the local network, causing the hostname command to return a loopback IP instead of a fully qualified domain name or IP address accessible by remote client machines. It is described in further detail at: ???

To remedy this issue you can set the *java.rmi.server.hostname* system property to control the hostname/ip reported to the RMI runtime when advertising the JMX ConnectorServer. This can also be used to dictate the address returned on a computer with multiple network interfaces to control reachability. To do so, add the value *-Djava.rmi.server.hostname=<desired hostname/ip>* to the QPID_OPTS environment variable before starting the *qpid-server* script.

1.1.5. Qpid JMX Management Console User Guide

1.1.5.1. Qpid JMX Management Console User Guide

The Qpid JMX Management Console is a standalone Eclipse RCP application for managing and monitoring the Qpid Java server utilising its JMX management interfaces.

This guide will give an overview of configuring the console, the features supported by it, and how to make use of the console in managing the various JMX Management Beans (MBeans) offered by the Qpid Java server.

1.1.5.2. Startup & Configuration

1.1.5.2.1. Startup

The console can be started in the following way, depending on platform:

- *Windows:* by running the *qpidmc.exe* executable file.
- *Linux:* by running the *qpidmc* executable.
- *Mac OS X:* by launching the *Qpid Management Console.app* application bundle.

1.1.5.2.2. SSL configuration

Newer Qpid Java servers can protect their JMX connections with SSL, and this is enabled by default. When attempting to connect to a server with this enabled, the console must be able to verify the SSL certificate presented to it by the server or the connection will fail.

If the server makes use of an SSL certificate signed by a known Signing CA (Certification Authority) then the console needs no extra configuration, and will make use of Java's default system-wide CA TrustStore for certificate verification (you may however have to update the system-wide default CA TrustStore if your certified is signed by a less common CA that is not already present in it).

If however the server is equipped with a self-signed SSL certificate, then the management console must be provided with an appropriate SSL TrustStore containing the public key for the SSL certificate, so that it

is able to validate it when presented by the server. The server ships with a script to create an example self-signed SSL certificate, and store the relevant entries in a KeyStore and matching TrustStore. This script can serve as a guide on how to use the Java Keytool security utility to manipulate your own stores, and more information can be found in the JSSE Reference Guide: http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CustomizingStores.

Supplying the necessary details to the console is performed by setting the *javax.net.ssl.trustStore* and *javax.net.ssl.trustStorePassword* environment variables when starting it. This can be done at the command line, but the preferred option is to set the configuration within the *qpidmc.ini* launcher configuration file for repeated usage. This file is equipped with a template to ease configuration, this should be uncommented and edited to suit your needs. It can be found in the root of the console releases for Windows, and Linux. For Mac OS X the file is located within the consoles *.app* application bundle, and to locate and edit it you must select *'Show Package Contents'* when accessing the context menu of the application, then browse to the *Contents/MacOS* sub folder to locate the file.

1.1.5.2.3. JMXMP configuration

Older releases of the Qpid Java server can make use of the Java Management Extensions Messaging Protocol (JMXMP) to provide protection for their JMX connections. This occurs when the server has its main configuration set with the management *'security-enabled'* property set to true.

In order to connect to this configuration of server, the console needs an additional library that is not included within the Java SE platform and cannot be distributed with the console due to licensing restrictions.

You can download the JMX Remote API 1.0.1_04 Reference Implementation from the Sun website ???. The included *jmxremote-1_0_1-bin/lib/jmxremote_optional.jar* file must be added to the *plugins/jmxremote.sasl_1.0.1* folder of the console release (again, in Mac OS X you will need to select *'Show package contents'* from the context menu whilst selecting the management console bundle in order to reveal the inner file tree).

Following this the console will automatically load the JMX Remote Optional classes and negotiate the SASL authentication profile type when encountering a JMXMP enabled Qpid Java server.

1.1.5.3. Managing Server Connections

1.1.5.3.1. Main Toolbar

The main toolbar of the console can be seen in the image below. The left most buttons respectively allow for adding a new server connection, reconnecting to an existing server selected in the connection tree, disconnecting the selected server connection, and removing the server from the connection tree.

Beside these buttons is a combo for selecting the refresh interval; that is, how often the console requests updated information to display for the currently open area in the main view. Finally, the right-most button enables an immediate update.

1.1.5.3.2. Connecting to a new server

To connect to a new server, press the *Add New Server* toolbar button, or select the *Qpid Manager -> Add New Connection* menu item. At this point a dialog box will be displayed requesting the server details, namely the server hostname, management port, and a username and password. An example is shown below:

Once all the required details are entered, pressing Connect will initiate a connection attempt to the server. It the attempt fails a reason will be shown and the server will not be added to the connection tree. If the attempt is successful the server will be added to the connections list and the entry expanded to show the initial administration MBeans the user has access to and any VirtualHosts present on the server, as can be seen in the figure below.

If the server supports a newer management API than the console in use, once connected this initial screen will contain a message on the right, indicating an upgraded console should be sought by the user to ensure all management functionality supported by the server is being utilised.

1.1.5.3.3. Reconnecting to a server

If a server has been connected to previously, it will be saved as an entry in the connection tree for further use. On subsequent connections the server can simply be selected from the tree and using the *Reconnect* toolbar button or *Qpid Manager -> Reconnect* menu item. At this stage the console will prompt simply for the username and password with which the user wishes to connect, and following a successful connection the screen will appear as shown previously above.

1.1.5.3.4. Disconnecting from a server

To disconnect from a server, select the connection tree node for the server and press the *Disconnect* toolbar button, or use the *Qpid Manager -> Disconnect* menu option.

1.1.5.3.5. Removing a server

To remove a server from the connection list, select the connection tree node for the server and press the *Remove* toolbar button, or use the *Qpid Manager -> Remove Connection* menu option.

1.1.5.4. Navigating a connected server

Once connected to a server, the various areas available for administration are accessed using the Qpid Connections tree at the left side of the application. To open a particular MBean from the tree for viewing, simply select it in the tree and it will be opened in the main view.

As there may be vast numbers of Queues, Connections, and Exchanges on the server these MBeans are not automatically added to the tree along with the general administration MBeans. Instead, dedicated selection areas are provided to allow users to select which Queue/Connection/Exchange they wish to view or add to the tree. These areas can be found by clicking on the Connections, Exchanges, and Queues nodes in the tree under each VirtualHost, as shown in the figure above. One or more MBeans may be selected and added to the tree as Favourites using the button provided. These settings are saved for future use, and each time the console connects to the server it will check for the presence of the MBean previously in the tree and add them if they are still present. Queue/Connection/Exchange MBeans can be removed from the tree by right clicking on them to expose a context menu allowing deletion.

As an alternative way to open a particular MBean for viewing, without first adding it to the tree, you can simply double click an entry in the table within the Queue/Connection/Exchange selection areas to open it immediately. It is also possible to open some MBeans like this whilst viewing certain other MBeans. When opening an MBean in either of these ways, a Back button is enabled in the top right corner of the

main view. Using this button will return you to the selection area or MBean you were previously viewing. The history resets each time the tree is used to open a new area or MBean.

1.1.5.5. ConfigurationManagement MBean

The ConfigurationManagement MBean is available on newer servers, to users with admin level management rights. It offers the ability to perform a live reload of the *Security* sections defined in the main server configuration file (e.g. defaults to: *etc/config.xml*). This is mainly to allow updating the server Firewall configuration to new settings without a restart, and can be performed by clicking the Execute button and confirming the prompt which follows.

1.1.5.6. LoggingManagement MBean

The LoggingManagement MBean is available on newer servers, and accessible by admin level users. It allows live alteration of the logging behaviour, both at a Runtime-only level and at the configuration file level. The latter can optionally affect the Runtime configuration, either through use of the servers automated LogWatch ability which detects changes to the configuration file and reloads it, or by manually requesting a reload. This functionality is split across two management tabs, Runtime Options and ConfigurationFile Options.

1.1.5.6.1. Runtime Options

The Runtime Options tab allows manipulation of the logging settings without affecting the configuration files (this means the changes will be lost when the server restarts), and gives individual access to every Logger active within the server.

As shown in the figure above, the table in this tab presents the Effective Level of each Logger. This is because the Loggers form a hierarchy in which those without an explicitly defined (in the logging configuration file) Level will inherit the Level of their immediate parent; that is, the Logger whose full name is a prefix of their own, or if none satisfy that condition then the RootLogger is their parent. As example, take the *org.apache.qpid* Logger. It is parent to all those below it which begin with *org.apache.qpid* and unless they have a specific Level of their own, they will inherit its Level. This can be seen in the figure, whereby all the children Loggers visible have a level of WARN just like their parent, but the RootLogger Level is INFO; the children have inherited the WARN level from *org.apache.qpid* rather than INFO from the RootLogger.

To aid with this distinction, the Logger Levels that are currently defined in the configuration file are highlighted in the List. Changing these levels at runtime will also change the Level of all their children which haven't been set their own Level using the runtime options. In the latest versions of the LoggingManagement MBean, it is possible to restore a child logger that has had an explicit level se, to inheriting that of its parent by setting it to an INHERITED level that removes any previously set Level of its own.

In order to set one of more Loggers to a new Level, they should be selected in the table (or double click an individual Logger to modify it) and the *Edit Selected Logger(s)* button pressed to load the dialog shown above. At this point, any of the available Levels supported by the server can be applied to the Loggers selected and they will immediately update, as will any child Loggers without their own specific Level.

The RootLogger can be similarly edited using the button at the bottom left of the window.

1.1.5.6.2. ConfigurationFile Options

The ConfigurationFile Options tab allows alteration of the Level settings for the Loggers defined in the configuration file, allowing changes to persist following a restart of the server. Changes made to the configuration file are only applied automatically while the sever is running if it was configured to enable the LogWatch capability, meaning it will monitor the configuration file for changes and apply the new configuration when the change is detected. If this was not enabled, the changes will be picked up when the server is restarted. The status of the LogWatch feature is shown at the bottom of the tab. Alternatively, in the latest versions of the LoggingManagement MBean it is possible to reload the logging configuration file on demand.

Manipulating the Levels is as on the Runtime Options tab, either double-click an individual Logger entry or select multiple Loggers and use the button to load the dialog to set the new Level.

One issue to note of when reloading the configuration file settings, either automatically using LogWatch or manually, is that any Logger set to a specific Level using the Runtime Options tab that is not defined in the configuration file will maintain that Level when the configuration file is reloaded. In other words, if a Logger is defined in the configuration file, then the configuration file will take precedence at reload, otherwise the Runtime options take precedence.

This situation will be immediately obvious by examining the Runtime Options tab to see the effective Level of each Logger – unless it has been altered with the RuntimeOptions or specifically set in the configuration file, a Logger Level should match that of its parent. In the latest versions of the LoggingManagement MBean, it is possible to use the RuntimeOptions to restore a child logger to inheriting from its parent by setting it with an INHERITED level that removes any previously set Level of its own.

1.1.5.7. ServerInformation MBean

The ServerInformation MBean currently only conveys various pieces of version information to allow precise identification of the server version and its management capabilities. In future it is likely to convey additional server-wide details and/or functionality.

1.1.5.8. UserManagement MBean

The UserManagement MBean is accessible by admin level users, and allows manipulation of existing user accounts and creation of new user accounts.

To add a new user, press the *Add New User* button, which will load the dialog shown below.

Here you may enter the new users Username, Password, and select their JMX Management Rights. This controls whether or not they have access to the management interface, and if so what capabilities are accessible. *Read Only* access allows undertaking any operations that do not alter the server state, such as viewing messages. *Read + Write* access allows use of all operations which are not deemed admin-only (such as those in the UserManagement MBean itself). *Admin* access allows a user to utilize any operation,

and view the admin-only MBeans (currently these are ConfigurationManagement, LoggingManagement, and UserManagement).

One or more users at a time may be deleted by selecting them in the table and clicking the *Delete User(s)* button. The console will then prompt for confirmation before undertaking the removals. Similarly, the access rights for one or more users may be updated by selecting them in the table and clicking the *Set Rights* button. The console will then display a dialog enabling selection of the new access level and confirmation to undertake the update.

An individual user password may be updated by selecting the user in the table in and clicking the *Set Password* button. The console will then display a dialog enabling input of the new password and confirmation to undertake the update.

The server caches the user details in memory to aid performance. If may sometimes be necessary to externally modify the password and access right files on disk. In order for these changes to be known to the server without a restart, it must be instructed to reload the file contents. This can be done using the provided *Reload User Details* button (on older servers, only the management rights file is reloaded, on newer servers both files are. The description on screen will indicate the behaviour). After pressing this button the console will seek confirmation before proceeding.

1.1.5.9. VirtualHostManager MBean

Each VirtualHost in the server has an associated VirtualHostManager MBean. This allows viewing, creation, and deletion of Queues and Exchanges within the VirtualHost.

Clicking the *Create* button in the Queue section will open a dialog allowing specification of the Name, Owner (optional), and durability properties of the new Queue, and confirmation of the operation.

One or more Queues may be deleted by selecting them in the table and clicking the *Delete* button. This will unregister the Queue bindings, remove the subscriptions and delete the Queue(s). The console will prompt for confirmation before undertaking the operation.

Clicking the *Create* button in the Exchange section will open a dialog allowing specification of the Name, Type, and Durable attributes of the new Exchange, and confirmation of the operation.

One or more Exchanges may be deleted by selecting them in the table and clicking the *Delete* button. This will unregister all the related channels and Queue bindings then delete the Exchange(s). The console will prompt for confirmation before undertaking the operation.

Double-clicking on a particular Queue or Exchange name in the tables will open the MBean representing it.

1.1.5.10. Notifications

MBeans on the server can potentially send Notifications that users may subscribe to. When managing an individual MBean that offers Notifications types for subscription, the console supplies a Notifications tab to allow (un)subscription to the Notifications if desired and viewing any that are received following subscription.

In order to provide quicker access to/awareness of any received Notifications, each VirtualHost area in the connection tree has a Notifications area that aggregates all received Notifications for MBeans in that VirtualHost. An example of this can be seen in the figure below.

All received Notifications will be displayed until such time as the user removes them, either in this aggregated view, or in the Notifications area of the individual MBean that generated the Notification.

They may be cleared selectively or all at once. To clear particular Notifications, they should be selected in the table before pressing the *Clear* button. To clear all Notifications, simply press the *Clear* button without anything selected in the table, at which point the console will request confirmation of this clear-all action.

1.1.5.11. Managing Queues

As mentioned in earlier discussion of Navigation, Queue MBeans can be opened either by double clicking an entry in the Queues selection area, or adding a queue to the tree as a favourite and clicking on its tree node. Unique to the Queue selection screen is the ability to view additional attributes beyond just that of the Queue Name. This is helpful for determining which Queues satisfy a particular condition, e.g. having <X> messages on the queue. The example below shows the selection view with additional attributes *Consumer Count, Durable, MessageCount, and QueueDepth* (selected using the *Select Attributes* button at the bottom right corner of the table).

Upon opening a Queue MBean, the Attributes tab is displayed, as shown below. This allows viewing the value all attributes, editing those which are writable values (highlighted in blue) if the users management permissions allow, viewing descriptions of their purpose, and graphing certain numerical attribute values as they change over time.

The next tab contains the operations that can be performed on the queue. The main table serves as a means of viewing the messages on the queue, and later for selecting specific messages to operate upon. It is possible to view any desired range of messages on the queue by specifying the visible range using the fields at the top and pressing the *Set* button. Next to this there are helper buttons to enable faster browsing through the messages on the queue; these allow moving forward and back by whatever number of messages is made visible by the viewing range set. The Queue Position column indicates the position of each message on the queue, but is only present when connected to newer servers as older versions cannot provide the necessary information to show this (unless only a single message position is requested).

Upon selecting a message in the table, its header properties and redelivery status are updated in the area below the table. Double clicking a message in the table (or using the *View Message Content* button to its right) will open a dialog window displaying the contents of the message.

One or more messages can be selected in the table and moved to another queue in the VirtualHost by using the *Move Message(s)* button, which opens a dialog to enable selection of the destination and confirmation of the operation. Newer servers support the ability to similarly copy the selected messages to another queue in a similar fashion, or delete the selected messages from the queue after prompting for confirmation.

Finally, all messages (that have not been acquired by consumers) on the queue can be deleted using the *Clear Queue* button, which will generate a prompt for confirmation. On newer servers, the status bar at the lower left of the application will report the number of messages actually removed.

1.1.5.12. Managing Exchanges

Exchange MBeans are opened for management operations in similar fashion as described for Queues, again showing an Attributes tab initially, with the Operations tab next:

Of the four default Exchange Types *(direct, fanout, headers, and topic)* all but *headers* have their bindings presented in the format shown above. The left table provides the binding/routing keys present in the exchange. Selecting one of these entries in the table prompts the right table to display all the queues associated with this key. Pressing the *Create* button opens a dialog allowing association of an existing queue with the entered Binding.

The *headers* Exchange type (default instantiation *amq.match or amq.headers*) is presented as below:

In the previous figure, the left table indicates the binding number, and the Queue associated with the binding. Selecting one of these entries in the table prompts the right table to display the header values that control when the binding matches an incoming message.

Pressing the *Create* button when managing a *headers* Exchange opens a dialog allowing creation of a new binding, associating an existing Queue with a particular set of header keys and values. The *x-match* key is required, and instructs the server whether to match the binding with incoming messages based on ANY or ALL of the further key-value pairs entered. If it is desired to enter more than 4 pairs, you may press the *Add additional field* button to create a new row as many times as is required. When managing a *headers* Exchange, double clicking an entry in the left-hand table will open the MBean for the Queue specified in the binding properties.

When managing another Exchange Type, double clicking the Queue Name in the right-hand table will open the MBean of the Queue specified.

1.1.5.13. Managing Connections

Exchange MBeans are opened for management operations in similar fashion as described for Queues, again showing an Attributes tab initially, with the Operations tab next, and finally a Notifications tab allowing subscription and viewing of Notifications. The Operations tab can be seen in the figure below.

The main table shows the properties of all the Channels that are present on the Connection, including whether they are *Transactional*, the *Number of Unacked Messages* on them, and the *Default Queue* if there is one (or *null* if there is not).

The main operations supported on a connection are Commiting and Rolling Back of Transactions on a particular Channel, if the Channel is Transactional. This can be done by selecting a particular Channel in the table and pressing the *Commit Transactions* or *Rollback Transactions* buttons at the lower right corner of the table, at which point the console will prompt for confirmation of the action. These buttons are only active when the selected Channel in the table is Transactional.

The final operation supported is closing the Connection. After pressing the *Close Connection* button, the console will prompt for confirmation of the action. If this is carried out, the MBean for the Connection being managed will be removed from the server. The console will be notified of this by the server and

display an information dialog to that effect, as it would if any other MBean were to be unregistered whilst being viewed.

Double clicking a row in the table will open the MBean of the associated *Default Queue* if there is one.

1.1.6. Qpid Management Features

Management tool: See our ??? for details of how to use various console options with the Qpid management features.

The management of QPID is categorised into following types-

1. Exchange
2. Queue
3. Connection
4. Broker

1) Managing and Monitoring Exchanges: Following is the list of features, which we can have available for managing and monitoring an Exchange running on a Qpid Server Domain-

1. Displaying the following information for monitoring purpose-
    a. The list of queues bound to the exchange along with the routing keys.
    b. General Exchange properties(like name, durable etc).
2. Binding an existing queue with the exchange.

2) Managing and Monitoring Queues: Following are the features, which we can have for a Queue on a Qpid Server Domain-

1. Displaying the following information about the queue for monitoring purpose-
    a. General Queue properties(like name, durable, etc.)
    b. The maximum size of a message that can be accepted from the message producer.
    c. The number of the active consumers accessing the Queue.
    d. The total number of consumers (Active and Suspended).
    e. The number of undelivered messages in the Queue.
    f. The total number of messages received on the Queue since startup.
    g. The maximum number of bytes for the Queue that can be stored on the Server.
    h. The maximum number of messages for the Queue that can be stored on the Server.
2. Viewing the messages on the Queue.
3. Deleting message from top of the Queue.
4. Clearing the Queue.

5. Browsing the DeadMessageQueue - Messages which are expired or undelivered because of some reason are routed to the DeadMessageQueue. This queue can not be deleted. [Note: The is open because it depends on how these kind of messages will be handeled?]

3) Managing and Monitoring Connections: Following are the features, which we can have for a connection on a QPID Server Domain-

1. Displaying general connection properties(like remote address, etc.).
2. Setting maximum number of channels allowed for a connection.
3. View all related channels and channel properties.
4. Closing a channel.
5. Commit or Rollback transactions of a channel, if the channel is transactional.
6. Notification for exceeding the maximum number of channels.
7. Dropping a connection.
8. The work for ??? implies that there are potentially some additional requirements
    a. Alert when tcp flow control kicks in
    b. Information available about current memory usage available through JMX interface
    c. Dynamic removal of buffer bounds? (fundamentally not possible with TransportIO)
    d. Management functionality added to JMX interface - UI changes?

4) Managing the Broker: Features for the Broker-

1. Creating an Exchange.
2. Unregistering an Exchange.
3. Creating a Queue.
4. Deleting a Queue.

Chapter 10. Management Tools

1. MessageStore Tool

1.1. MessageStore Tool

We have a number of implementations of the Qpid MessageStore interface. This tool allows the interrogation of these stores while the broker is offline.

1.1.1. MessageStore Implementations

- ???
- ???
- ???

1.1.2. Introduction

Each of the MessageStore implementations provide different back end storage for their messages and so would need a different tool to be able to interrogate their contents at the back end.

What this tool does is to utilise the Java broker code base to access the contents of the storage providing the user with a consistent means to inspect the storage contents in broker memory. The tool allows the current messages in the store to be inspected and copied/moved between queues. The tool uses the message instance in memory for all its access paths, but changes made will be reflected in the physical store (if one exists).

1.1.3. Usage

The tools-distribution currently includes a unix shell command 'msTool.sh' this script will launch the java tool.

The tool loads $QPID_HOME/etc/config.xml by default. If an alternative broker configuration is required this should be provided on the command line as would be done for the broker.

```
msTool.sh -c <path to different config.xml>
```

On startup the user is present with a command prompt

```
$ msTool.sh
MessageStoreTool - for examining Persistent Qpid Broker MessageStore instances
bdb$
```

1.1.4. Available Commands

The available commands in the tool can be seen through the use of the 'help' command.

```
bdb$ help
+-----------------------------------------------------------------+
```

```
|                       Available Commands                        |
+-----------------------------------------------------------------+
| Command | Description                                           |
+-----------------------------------------------------------------+
| quit    | Quit the tool.                                        |
| list    | list available items.                                 |
| dump    | Dump selected message content. Default: show=content  |
| load    | Loads specified broker configuration file.            |
| clear   | Clears any selection.                                 |
| show    | Shows the messages headers.                           |
| select  | Perform a selection.                                  |
| help    | Provides detailed help on commands.                   |
+-----------------------------------------------------------------+
bdb$
```

A brief description is displayed and further usage information is shown with 'help <command>'

```
bdb$ help list
list availble items.
Usage:list queues [<exchange>] | exchanges | bindings [<exchange>] | all
bdb$
```

1.1.5. Future Work

Currently the tool only works whilst the broker is offline i.e. it is up, but not accepting AMQP connections. This requires a stop/start of the broker. If this functionality was incorporated into the broker then a telnet functionality could be provided allowing online management.

2. Qpid Java Broker Management CLI

2.1. How to build Apache Qpid CLI

2.1.1. Build Instructions - General

At the very beginning please build Apache Qpid by refering this installation guide from here ???.

After successfully build Apache Qpid you'll be able to start Apache Qpid Java broker,then only you are in a position to use Qpid CLI.

2.1.2. Check out the Source

First check out the source from subversion repository. Please visit the following link for more information about different versions of Qpid CLI.

???

2.1.3. Prerequisites

For the broker code you need JDK 1.5.0_15 or later. You should set JAVA_HOME and include the bin directory in your PATH.

Check it's ok by executing java -v !

2.1.4. Building Apache Qpid CLI

This project is currently having only an ant build system.Please install ant build system before trying to install Qpid CLI.

2.1.5. Compiling

To compile the source please run following command

```
ant compile
```

To compile the test source run the following command

```
ant compile-tests
```

2.1.6. Running CLI

After successful compilation set QPID_CLI environment variable to the main source directory.(set the environment variable to the directory where ant build script stored in the SVN checkout).Please check whether the Qpid Java broker is up an running in the appropriate location and run the following command to start the Qpid CLI by running the qpid-cli script in the bin directory.

$QPID_CLI/bin/qpid-cli -h <hostname of the broker> -p <broker running port> For more details please have a look in to README file which ships with source package of Qpid CLI.

2.1.7. Other ant targets

For now we are supporting those ant targets.

ant clean	Clean the complete build including CLI build and test build.
ant jar	Create the jar file for the project without test cases.
ant init	Create the directory structure for build.
ant compile-tests	This compiles all the test source.
ant test	Run all the test cases.

Part IV. AMQP Messaging Clients Clients

Table of Contents

Chapter 11. AMQP Java JMS Messaging Client

The Java Client supported by Qpid implements the Java JMS 1.1 Specification [http://java.sun.com/products/jms/docs.html].

1. General User Guides

1.1. System Properties

1.1.1. Explanation of System properties used in Qpid

This page documents the various System Properties that are currently used in the Qpid Java code base.

1.1.1.1. Client Properties

STRICT_AMQP	Type	Boolean
	Default	FALSE
	This forces the client to only send AMQP compliant frames. This will disable a number of JMS features. **Features disabled by STRICT_AMQP** • Queue Browser • Message Selectors • Durable Subscriptions • Session Recover may result in duplicate message delivery • Destination validation, so no InvalidDestinationException will be thrown This is associated with property STRICT_AMQP_FATAL	
STRICT_AMQP_FATAL	Type	Boolean
	Default	FALSE
	This will cause any attempt to utilise an enhanced feature to throw and UnsupportedOperationException. When set	

	to false then the exception will not occur but the feature will be disabled. e.g. The Queue Browser will always show no messages. Any message selector will be removed.	
IMMEDIATE_PREFETCH	Type	Boolean
	Default	FALSE
	The default with AMQP is to start prefetching messages. However, with certain 3rd party Java tools, such as Mule this can cause a problem. Mule will create a consumer but never consume from it so any any prefetched messages will be stuck until that session is closed. This property is used to re-instate the default AMQP behaviour. The default Qpid behaviour is to prevent prefetch occurring, by starting the connection Flow Controlled, until a request for a message is made on the consumer either via a receive() or setting a message listener.	
amqj.default_syncwrite_timeout	Type	long
	Default	30000
	The number length of time in millisecond to wait for a synchronous write to complete.	
amq.dynamicsaslregistrar.properties	Type	String
	Default	org/apache/qpid/client/security/DynamicSaslRegistrar.properties
	The name of the SASL configuration properties file.	
amqj.heartbeat.timeoutFactor	Type	float
	Default	2.0
	The factor used to get the timeout from the delay between heartbeats	
amqj.tcpNoDelay	Type	Boolean
	Default	TRUE
	Disable Nagle's algorithm on the TCP connection.	
amqj.sendBufferSize	integer	Boolean

	Default	32768
	This is the default buffer sized created by Mina.	
amqj.receiveBufferSize	Type	integer
	Default	32768
	This is the default buffer sized created by Mina.	
amqj.protocolprovider.class	Type	String
	Default	org.apache.qpid.server.protocol.AMQPFastPı
	This specifies the default IoHandlerAdapter that represents the InVM broker. The IoHandlerAdapter must have a constructor that takes a single Integer that represents the InVM port number.	
amqj.protocol.logging.level	Type	Boolean
	Default	null
	If set this will turn on protocol logging on the client.	
jboss.host	Used by the JBossConnectionFactoryInitialiser to specify the host to connect to perform JNDI lookups.	
jboss.port	Used by the JBossConnectionFactoryInitialiser to specify the port to connect to perform JNDI lookups.	
amqj.MaximumStateWait	Default	30000
	Used to set the maximum time the State Manager should wait before timing out a frame wait.	

1.1.1.2. Management Properties

security	Default	null
	String representing the Security level to be used to on the connection to the broker. The null default results in no security or PLAIN. When used with jmxconnector 'javax.management.remote.jmxmp.JMXMPConnector' a security value of 'CRAM-MD5' will result in all communication to the broker being encrypted.	
jmxconnector	Default	null
	String representing the JMXConnector class used to perform the connection to the broker. The null default results in the standard JMX	

connector. Utilising 'javax.management.remote.jmxmp.JMXMPConnector' and security 'CRAM-MD5' will result in all communication to the broker being encrypted.

timeout — Default 5000

Long value representing the milli seconds before connection to the broker should timeout.

1.1.1.3. Properties used in Examples

archivepath — Used in `FileMessageDispatcher`. This properties specifies the directory to move payload file(s) to archive location as no error

1.2. Connection URL Format

1.2.1. Format

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

The connection url defines the values that are common across the cluster of brokers. The virtual host is second in the list as the AMQP specification demands that it start with a '/' otherwise it be more readable to be swapped with clientid. There is currently only one required option and that is the *brokerlist* option. In addition the following options are recognised.

1.2.2. Worked Example

You could use a URL which looks something like this:

```
amqp://guest:guest@client1/development?brokerlist='tcp://localhost:5672'
```

Breaking this example down, here's what it all means:

- amqp = the protocol we're using
- guest:guest@localhost = username:password@clientid where the clientid is the name of your server (used under the covers but don't worry about this for now). Always use the guest:guest combination at the moment.
- development = the name of the virtualhost, where the virtualhost is a path which acts as a namespace. You can effectively use any value here so long as you're consistent throughout. The virtualhost must start with a slash "/" and continue with names separated by slashes. A name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_+!=:].
- brokerlist = this is the host address and port for the broker you want to connect to. The connection factory will assume tcp if you don't specify a transport protocol. The port also defaults to 5672. Naturally you have to put at least one broker in this list.

This example is not using failover so only provides one host for the broker. If you do wish to connect using failover you can provide two (or more) brokers in the format:

brokerlist='tcp://host1&tcp://host2:5673'

The default failover setup will automatically retry each broker once after a failed connection. If the brokerlist contains more than one server then these servers are tried in a round robin. Details on how to modifiy this behaviour will follow soon !

1.2.3. Options

Table 11.1. Connection URL Options

Option	Default	Description
brokerlist	see below	The list of brokers to use for this connection
failover	see below	The type of failover method to use with the broker list.
maxprefetch	5000	The maximum number of messages to prefetch from the broker.

1.2.4. Brokerlist option

```
brokerlist='<broker url>[;<broker url>]'
```

The broker list defines the various brokers that can be used for this connection. A minimum of one broker url is required additional URLs are semi-colon(';') delimited.

1.2.5. Broker URL format

```
<transport>://<host>[:<port>][?<option>='<value>'[&<option>='<value>']]
```

There are currently quite a few default values that can be assumed. This was done so that the current client examples would not have to be re-written. The result is if there is no transport, 'tcp' is assumed and the default AMQP port of 5672 is used if no port is specified.

Table 11.2. Broker URL- Transport

Transport
tcp
vm

Currently only 'tcp' and 'vm' transports are supported. Each broker can take have additional options that are specific to that broker. The following are currently implemented options. To add support for further transports the "client.transportTransportConnection" class needs updating along with the parsing to handle the transport.

Table 11.3. Broker URL - Connection Options

Option	Default	Description

retries	1	The number of times to retry connection to this Broker
ssl	false	Use ssl on the connection
connecttimeout	30000	How long in (milliseconds) to wait for the connection to succeed
connectdelay	none	How long in (milliseconds) to wait before attempting to reconnect

1.2.6. Brokerlist failover option

```
failover='<method>[?<options>]'
```

This option controls how failover occurs when presented with a list of brokers. There are only two methods currently implemented but interface qpid.jms.failover.FailoverMethod can be used for defining further methods.

Currently implemented failover methods.

Table 11.4. Broker List - Failover Options

Method	Description
singlebroker	This will only use the first broker in the list.
roundrobin	This method tries each broker in turn.
nofailover	[New in 0.5] This method disables all retry and failover logic.

The current defaults are naturally to use the 'singlebroker' when only one broker is present and the 'roundrobin' method with multiple brokers. The '"method"' value in the URL may also be any valid class on the classpath that implements the FailoverMethod interface.

The 'nofailover' method is useful if you are using a 3rd party tool such as Mule that has its own reconnection strategy that you wish to use.

Table 11.5. Broker List - Failover Options

Option	Default	Description
cyclecount	1	The number of times to loop through the list of available brokers before failure.

Note: Default was changed from 0 to 1 in Release 0.5

1.2.7. Sample URLs

```
amqp:///test?brokerlist='localhost'
amqp:///test?brokerlist='tcp://anotherhost:5684?retries='10''
amqp://guest:guest@/test?brokerlist='vm://:1;vm://:2'&failover='roundrobin'
```

```
amqp://guest:guest@/test?brokerlist='vm://:1;vm://:2'&failover='roundrobin?cycleco
amqp://guest:guest@client/test?brokerlist='tcp://localhost;tcp://redundant-server:
amqp://guest:guest@/test?brokerlist='vm://:1'&failover='nofailover'
```

1.3. Binding URL Format

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

This URL format is used for two purposes in the code base. The broker uses this in the XML configuration file to create and bind queues at broker startup. It is also used by the client as a destination.

This format was used because it allows an explicit description of exchange and queue relationship.

The Exchange Class is not normally required for client connection as clients only publish to a named exchange however if exchanges are being dynamically instantiated it will be required. The class is required for the server to instantiate an exchange.

There are a number of options that are currently defined:

Table 11.6. Binding URL Options

Option	type	Description
exclusive	boolean	Is this an exclusive connection
autodelete	boolean	Should this queue be deleted on client disconnection
durable	boolean	Create a durable queue
clientid	string	Use the following client id
subscription	boolean	Create a subscription to this destination
routingkey	string	Use this value as the routing key

Using these options in conjunction with the Binding URL format should allow future expansion as new and custom exchange types are created.

The URL format requires *that at least one* Queue or routingkey option be present on the URL.

The routingkey would be used to encode a topic as shown in the examples section below.

1.3.1. Examples

Example 11.1. Queues

A queue can be created in QPID using the following URL format.

direct://amq.direct//<Queue Name>

For example: direct://amq.direct//simpleQueue

Queue names may consist of any mixture of digits, letters, and underscores.

Example 11.2. Topics

A topic can be created in QPID using the following URL format.

topic://amq.topic/<Topic Subscription>/

The topic subscription may only contain the letters A-Z and a-z and digits 0-9.

```
direct://amq.direct/SimpleQueue
direct://amq.direct/UnusuallyBoundQueue?routingkey='/queue'
topic://amq.topic?routingkey='stocks.#'
topic://amq.topic?routingkey='stocks.nyse.ibm'
```

1.4. Java JMS Selector Syntax

The AMQP Java JMS Messaging Client supports the following syntax for JMS selectors.

```
Comments:

  LINE_COMMENT: "--" (~["\n","\r"])* EOL
  EOL: "\n"|"\r"|"\r\n"
  BLOCK_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/"

Reserved Words (case insensitive):

  NOT:     "NOT"
  AND:     "AND"
  OR:      "OR"
  BETWEEN: "BETWEEN"
  LIKE:    "LIKE"
  ESCAPE:  "ESCAPE"
  IN:      "IN"
  IS:      "IS"
  TRUE:    "TRUE"
  FALSE:   "FALSE"
  NULL:    "NULL"

Literals (case insensitive):

  DECIMAL_LITERAL:        ["1"-"9"] (["0"-"9"])* (["l","L"])?
  HEX_LITERAL:            "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+
  OCTAL_LITERAL:          "0" (["0"-"7"])*
  FLOATING_POINT_LITERAL: (   (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? // match
                            | "." (["0"-"9"])+ (<EXPONENT>)?              // match
                            | (["0"-"9"])+ <EXPONENT> )                   // match
  EXPONENT:               "E" (["+","-"])? (["0"-"9"])+
  STRING_LITERAL:         "'" ( ("''") | ~["'"] )*  "'"

Identifiers (case insensitive):

  ID : ["a"-"z", "_", "$"] (["a"-"z","0"-"9","_", "$"])*
  QUOTED_ID : "\"" ( ("\"\"") | ~["\""] )*  "\""
```

```
Grammar:

  JmsSelector          := orExpression
  orExpression         := ( andExpression ( <OR> andExpression )* )
  andExpression        := ( equalityExpression ( <AND> equalityExpression )* )
  equalityExpression   := ( comparisonExpression (   "=" comparisonExpression
                                                   | "<>" comparisonExpression
                                                   | <IS> <NULL>
                                                   | <IS> <NOT> <NULL> )* )
  comparisonExpression := ( addExpression (   ">" addExpression
                                            | ">=" addExpression
                                            | "<" addExpression
                                            | "<=" addExpression
                                            | <LIKE> stringLitteral ( <ESCAPE> str
                                            | <NOT> <LIKE> <STRING_LITERAL> ( <ESC
                                            | <BETWEEN> addExpression <AND> addExp
                                            | <NOT> <BETWEEN> addExpression <AND>
                                            | <IN> "(" <STRING_LITERAL> ( "," <STR
                                            | <NOT> <IN> "(" <STRING_LITERAL> ( ",
  addExpression        := multExpr ( ( "+" multExpr | "-" multExpr ) )*
  multExpr             := unaryExpr ( "*" unaryExpr | "/" unaryExpr | "%" unaryExp
  unaryExpr            := ( "+" unaryExpr | "-" unaryExpr | <NOT> unaryExpr | prim
  primaryExpr          := ( literal | variable | "(" orExpression ")" )
  literal              := (   <STRING_LITERAL>
                            | <DECIMAL_LITERAL>
                            | <HEX_LITERAL>
                            | <OCTAL_LITERAL>
                            | <FLOATING_POINT_LITERAL>
                            | <TRUE>
                            | <FALSE>
                            | <NULL> )
  variable             := ( <ID> | <QUOTED_ID> )
```

2. AMQP Java JMS Examples

- Examples Directory [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/]
- Script for running examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/runSample.sh]
- Direct Example [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/org/apache/qpid/example/jmsexample/direct/]
- Fanout Example [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/org/apache/qpid/example/jmsexample/fanout/]
- Pub-Sub Example [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/org/apache/qpid/example/jmsexample/pubsub]
- Request/Response Example [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/org/apache/qpid/example/jmsexample/requestResponse/]
- Transacted Example [https://svn.apache.org/repos/asf/qpid/trunk/qpid/java/client/example/src/main/java/org/apache/qpid/example/jmsexample/transacted/]

Chapter 12. AMQP C++ Messaging Client

1. User Guides

- C++ Client API (AMQP 0-10) [http://qpid.apache.org/docs/api/cpp/html/index.html]

2. Examples

- AMQP C++ Client Examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/cpp/examples/]
- Running the AMQP C++ Client Examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/cpp/examples/README.txt]

Chapter 13. AMQP .NET Messaging Client

Currently the .NET code base provides two client libraries that are compatible respectively with AMQP 0.8 and 0.10. The 0.8 client is located in `qpid\dotnet` and the 0.10 client in: `qpid\dotnet \client-010`.

You will need an AMQP broker to fully use those client libraries. Use M4 or later C++ broker for AMQP 0.10 or Java broker for AMQP 0.8/0.9.

1. User Guides

1.1. Apache Qpid: Open Source AMQP Messaging - .NET User Guide

1.1.1. Tutorial

This tutorial consists of a series of examples using the three most commonly used exchange types - Direct, Fanout and Topic exchanges. These examples show how to write applications that use the most common messaging paradigms.

- direct

  In the direct examples, a message producer writes to the direct exchange, specifying a routing key. A message consumer reads messages from a named queue. This illustrates clean separation of concerns - message producers need to know only the exchange and the routing key, message consumers need to know only which queue to use on the broker.

- fanout

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

- pub-sub

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

- request-response

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

1.1.1.1. Running the Examples

Before running the examples, you need to unzip the file Qpid.NET-net-2.0-M4.zip, the following tree is created:

```
<home>
  |-qpid
     |-lib (contains the required dlls)
     |-examples
          |- direct
          |    |-example-direct-Listener.exe
          |    |-example-direct-Producer.exe
          |- fanout
          |    |-example-fanout-Listener.exe
          |    |-example-fanout-Producer.exe
          |- pub-sub
          |    |-example-pub-sub-Listener.exe
          |    |-example-pub-sub-Publisher.exe
          |- request-response
               |-example-request-response-Client.exe
               |-example-request-response-Server.exe
```

Make sure your PATH contains the directory <home>/qpid/lib The examples can be run by executing the provided exe files:

```
$ cd <home>/qpid/examples/examplefolder
$ example-...-.exe [hostname] [portnumber]
```

where [hostname] is the qpid broker host name (default is localhost) and [portnumber] is the port number on which the qpid broker is accepting connection (default is 5672).

1.1.1.2. Creating and Closing Sessions

All of the examples have been written using the Apache Qpid .NEt 0.10 API. The examples use the same skeleton code to initialize the program, create a session, and clean up before exiting:

```
using System;
using System.IO;
using System.Text;
using System.Threading;
using org.apache.qpid.client;
using org.apache.qpid.transport;

...

        private static void Main(string[] args)
        {
            string host = args.Length > 0 ? args[0] : "localhost";
            int port = args.Length > 1 ? Convert.ToInt32(args[1]) : 5672;
            Client connection = new Client();
            try
            {
                connection.connect(host, port, "test", "guest", "guest");
                ClientSession session = connection.createSession(50000);
```

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

                connection.close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Error: \n" + e.StackTrace);
            }
        }
...
```

1.1.1.3. Writing Direct Applications

This section describes two programs that implement direct messaging using a Direct exchange: • org.apache.qpid.example.direct.Producer (from example-direct-producer) publishes messages to the amq.direct exchange, using the routing key routing_key. •org.apache.qpid.example.direct.Listener (from example-direct-Listener) uses a message listener to receive messages from the queue named message_queue.

1.1.1.3.1. Running the Direct Examples

1) Make sure your PATH contains the directory <home>/qpid/lib

2) Make sure that a qpid broker is running:

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

If a broker is running, you should see the qpidd process in the output of the above command.

3) Read the messages from the message queue using direct listener, as follows:

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-direct-Listener.exe [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-direct-Listener.exe [hostname] [portnumber]
```

This program is waiting for messages to be published, see next step:

4) Publish a series of messages to the amq.direct exchange by running direct producer, as follows:

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-direct-Producer.exe  [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-direct-Producer.exe [hostname] [portnumber]
```

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

5) Go to the windows where you are running your listener. You should see the following output:

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

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

1.1.1.3.2. Reading Messages from the Queue

The program , listener.cs, is a message listener that receives messages from a queue.

First it creates a queue named message_queue, then binds it to the amq.direct exchange using the binding key routing_key.

```
//--------- Main body of program ------------------------------------------
// Create a queue named "message_queue", and route all messages whose
// routing key is "routing_key" to this newly created queue.
session.queueDeclare("message_queue");
session.exchangeBind("message_queue", "amq.direct", "routing_key");
```

The queue created by this program continues to exist after the program exits, and any message whose routing key matches the key specified in the binding will be routed to the corresponding queue by the broker. Note that the queue could have been be deleted using the following code:

```
session.queueDelete("message_queue");
```

To create a message listener, create a class derived from IMessageListener, and override the messageTransfer method, providing the code that should be executed when a message is received.

```
public class MessageListener : IMessageListener
{
  ......
  public void messageTransfer(IMessage m)
  {
  .....
}
```

The main body of the program creates a listener for the subscription; attaches the listener to a message queue; and subscribe to the queue to receive messages from the queue.

```
lock (session)
{
  // Create a listener and subscribe it to the queue named "message_queue"
  IMessageListener listener = new MessageListener(session);
  session.attachMessageListener(listener, "message_queue");
  session.messageSubscribe("message_queue");
  // Receive messages until all messages are received
  Monitor.Wait(session);
}
```

The MessageListener's messageTransfer() function is called whenever a message is received. In this example the message is printed and tested to see if it is the final message. Once the final message is received, the messages are acknowledged.

```
BinaryReader reader = new BinaryReader(m.Body, Encoding.UTF8);
byte[] body = new byte[m.Body.Length - m.Body.Position];
reader.Read(body, 0, body.Length);
ASCIIEncoding enc = new ASCIIEncoding();
string message = enc.GetString(body);
 Console.WriteLine("Message: " + message);
// Add this message to the list of message to be acknowledged
_range.add(m.Id);
if( message.Equals("That's all, folks!") )
{
  // Acknowledge all the received messages
  _session.messageAccept(_range);
  lock(_session)
  {
      Monitor.Pulse(_session);
  }
}
```

1.1.1.3.3. Publishing Messages to a Direct Exchange

The second program in the direct example, Producer.cs, publishes messages to the amq.direct exchange using the routing key routing_key.

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

```
IMessage message = new Message();
// The routing key is a message property. We will use the same
// routing key for each message, so we'll set this property
// just once. (In most simple cases, there is no need to set
// other message properties.)
message.DeliveryProperties.setRoutingKey("routing_key");
```

Now send some messages:

```
// Asynchronous transfer sends messages as quickly as
// possible without waiting for confirmation.
for (int i = 0; i < 10; i++)
{
  message.clearData();
  message.appendData(Encoding.UTF8.GetBytes("Message " + i));
  session.messageTransfer("amq.direct", message);
}
```

Send a final synchronous message to indicate termination:

```
// And send a syncrhonous final message to indicate termination.
message.clearData();
message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));
session.messageTransfer("amq.direct", "routing_key", message);
session.sync();
```

1.1.1.4. Writing Fanout Applications

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

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

1.1.1.4.1. Running the Fanout Examples

1) Make sure your PATH contains the directory <home>/qpid/lib

2) Make sure that a qpid broker is running:

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

If a broker is running, you should see the qpidd process in the output of the above command.

3) In separate windows, start one or more fanout listeners as follows:

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-fanout-Listener.exe [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-fanout-Listener.exe [hostname] [portnumber]
```

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

```
Listening
```

This program is waiting for messages to be published, see next step:

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

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-fanout-Producer.exe  [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-fanout-Producer.exe [hostname] [portnumber]
```

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

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

```
Message: Message 0
Message: Message 1
Message: Message 2
Message: Message 3
```

```
Message: Message 4
Message: Message 5
Message: Message 6
Message: Message 7
Message: Message 8
Message: Message 9
Message: That's all, folks!
```

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

1.1.1.5. Consuming from a Fanout Exchange

The first program in the fanout example, Listener.cs, creates a private queue, binds it to the amq.fanout exchange, and waits for messages to arrive on the queue, printing them out as they arrive. It uses a Listener that is identical to the one used in the direct example:

```
public class MessageListener : IMessageListener
  {
      private readonly ClientSession _session;
      private readonly RangeSet _range = new RangeSet();
      public MessageListener(ClientSession session)
      {
          _session = session;
      }

      public void messageTransfer(IMessage m)
      {
          BinaryReader reader = new BinaryReader(m.Body, Encoding.UTF8);
          byte[] body = new byte[m.Body.Length - m.Body.Position];
          reader.Read(body, 0, body.Length);
          ASCIIEncoding enc = new ASCIIEncoding();
          string message = enc.GetString(body);
          Console.WriteLine("Message: " + message);
          // Add this message to the list of message to be acknowledged
          _range.add(m.Id);
          if (message.Equals("That's all, folks!"))
          {
              // Acknowledge all the received messages
              _session.messageAccept(_range);
              lock (_session)
              {
                  Monitor.Pulse(_session);
              }
          }
      }
  }
```

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

```
string myQueue = session.Name;
```

```
session.queueDeclare(myQueue, Option.EXCLUSIVE, Option.AUTO_DELETE);
session.exchangeBind(myQueue, "amq.fanout", "my-key");
```

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

```
lock (session)
{
   Console.WriteLine("Listening");
   // Create a listener and subscribe it to my queue.
   IMessageListener listener = new MessageListener(session);
   session.attachMessageListener(listener, myQueue);
   session.messageSubscribe(myQueue);
   // Receive messages until all messages are received
   Monitor.Wait(session);
}
```

1.1.1.5.1. Publishing Messages to the Fanout Exchange

The second program in this example, Producer.cs, writes messages to the fanout queue.

```
// Unlike topic exchanges and direct exchanges, a fanout
// exchange need not set a routing key.
IMessage message = new Message();
// Asynchronous transfer sends messages as quickly as
// possible without waiting for confirmation.
for (int i = 0; i < 10; i++)
{
    message.clearData();
    message.appendData(Encoding.UTF8.GetBytes("Message " + i));
    session.messageTransfer("amq.fanout", message);
}

// And send a syncrhonous final message to indicate termination.
message.clearData();
message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));
session.messageTransfer("amq.fanout", message);
session.sync();
```

1.1.1.6. Writing Publish/Subscribe Applications

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

• Publisher.cS sends messages to the amq.topic exchange, using the multipart routing keys usa.news, usa.weather, europe.news, and europe.weather. • Listener.cs creates private queues for news, weather, usa, and europe, binding them to the amq.topic exchange using bindings that match the corresponding parts of the multipart routing keys.

In this example, the publisher creates messages for topics like news, weather, and sports that happen in regions like Europe, Asia, or the United States. A given consumer may be interested in all weather messages, regardless of region, or it may be interested in news and weather for the United States, but

uninterested in items for other regions. In this example, each consumer sets up its own private queues, which receive precisely the messages that particular consumer is interested in.

1.1.1.6.1. Running the Publish-Subscribe Examples

1) Make sure your PATH contains the directory <home>/qpid/lib

2) Make sure that a qpid broker is running:

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

If a broker is running, you should see the qpidd process in the output of the above command.

3) In separate windows, start one or more topic subscribers as follows:

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-pub-sub--Listener.exe [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-pub-sub-Listener.exe [hostname] [portnumber]
```

You will see output similar to this:

```
Listening for messages ...
Declaring queue: usa
Declaring queue: europe
Declaring queue: news
Declaring queue: weather
```

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

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

```
$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-pub-sub-Producer.exe  [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-pub-sub-Producer.exe [hostname] [portnumber]
```

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

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

```
Message: Message 0 from usa
Message: Message 0 from news
Message: Message 0 from weather
Message: Message 1 from usa
Message: Message 1 from news
Message: Message 2 from usa
Message: Message 2 from news
Message: Message 3 from usa
Message: Message 3 from news
Message: Message 4 from usa
Message: Message 4 from news
Message: Message 5 from usa
Message: Message 5 from news
Message: Message 6 from usa
Message: Message 6 from news
Message: Message 7 from usa
Message: Message 7 from news
Message: Message 8 from usa
Message: Message 8 from news
Message: Message 9 from usa
....
Message: That's all, folks! from weather
Shutting down listener for control
Message: That's all, folks! from europe
Shutting down listener for control
```

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

1.1.1.6.2. Publishing Messages to a Topic Exchange

The first program in the publish/subscribe example, Publisher.cs, defines two new functions: one that publishes messages to the topic exchange, and one that indicates that no more messages are coming.

The publishMessages function publishes a series of five messages using the specified routing key.

```
private static void publishMessages(ClientSession session, string routing_key)
{
 IMessage message = new Message();
 // Asynchronous transfer sends messages as quickly as
```

```
 // possible without waiting for confirmation.
 for (int i = 0; i < 10; i++)
 {
    message.clearData();
    message.appendData(Encoding.UTF8.GetBytes("Message " + i));
    session.messageTransfer("amq.topic", routing_key, message);
 }
}
```

The noMoreMessages function signals the end of messages using the control routing key, which is reserved for control messages.

```
private static void noMoreMessages(ClientSession session)
{
  IMessage message = new Message();
  // And send a syncrhonous final message to indicate termination.
  message.clearData();
  message.appendData(Encoding.UTF8.GetBytes("That's all, folks!"));
  session.messageTransfer("amq.topic", "control", message);
  session.sync();
}
```

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

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

noMoreMessages(session);
```

1.1.1.6.3. Reading Messages from the Queue

The second program in the publish/subscribe example, Listener.cs, creates a local private queue, with a unique name, for each of the four binding keys it specifies: usa.#, europe.#, #.news, and #.weather, and creates a listener.

```
Console.WriteLine("Listening for messages ...");
// Create a listener
prepareQueue("usa", "usa.#", session);
prepareQueue("europe", "europe.#", session);
prepareQueue("news", "#.news", session);
prepareQueue("weather", "#.weather", session);
```

The prepareQueue() method creates a queue using a queue name and a routing key supplied as arguments it then attaches a listener with the session for the created queue and subscribe for this receiving messages from the queue:

```
// Create a unique queue name for this consumer by concatenating
// the queue name parameter with the Session ID.
Console.WriteLine("Declaring queue: " + queue);
session.queueDeclare(queue, Option.EXCLUSIVE, Option.AUTO_DELETE);

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

session.exchangeBind(queue, "amq.topic", routing_key);
session.exchangeBind(queue, "amq.topic", "control");

// subscribe the listener to the queue
IMessageListener listener = new MessageListener(session);
session.attachMessageListener(listener, queue);
session.messageSubscribe(queue);
```

1.1.1.7. Writing Request/Response Applications

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

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

1.1.1.7.1. Running the Request/Response Examples

1) Make sure your PATH contains the directory <home>/qpid/lib

2) Make sure that a qpid broker is running:

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

If a broker is running, you should see the qpidd process in the output of the above command.

3) Run the server.

$ cd <home>/qpid/examples/direct

```
With cygwin:
```

$ ./example-request-response-Server.exe [hostname] [portnumber]

```
or with mono:
```

$ mono ./example-request-response-Server.exe [hostname] [portnumber]

```
      You will see output similar to this:
```

Waiting for requests

```
4) In a separate window, start a client:

$ cd <home>/qpid/examples/direct
```

With cygwin:

```
$ ./example-request-response-Client.exe [hostname] [portnumber]
```

or with mono:

```
$ mono ./example-request-response-Client.exe [hostname] [portnumber]
```

You will see output similar to this:

```
Activating response queue listener for: clientSystem.Byte[]
Waiting for all responses to arrive ...
Response: TWAS BRILLIG, AND THE SLITHY TOVES
Response: DID GIRE AND GYMBLE IN THE WABE.
Response: ALL MIMSY WERE THE BOROGROVES,
Response: AND THE MOME RATHS OUTGRABE.
Shutting down listener for clientSystem.Byte[]
Response: THAT'S ALL, FOLKS!
```

4) Go back to the server window, the output should be similar to this:

```
Waiting for requests
Request: Twas brillig, and the slithy toves
Request: Did gire and gymble in the wabe.
Request: All mimsy were the borogroves,
Request: And the mome raths outgrabe.
Request: That's all, folks!
```

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

1.1.1.7.2. The Client Application

The first program in the request-response example, Client.cs, sets up a private response queue to receive responses from the server, then sends messages the server, listening to the response queue for the server's responses.

```
string response_queue = "client" + session.getName();
// Use the name of the response queue as the routing key
session.queueDeclare(response_queue);
session.exchangeBind(response_queue, "amq.direct", response_queue);

// Create a listener for the response queue and listen for response messages.
Console.WriteLine("Activating response queue listener for: " + response_queue);
IMessageListener listener = new ClientMessageListener(session);
session.attachMessageListener(listener, response_queue);
session.messageSubscribe(response_queue);
```

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

```
IMessage request = new Message();
request.DeliveryProperties.setRoutingKey("request");
request.MessageProperties.setReplyTo(new ReplyTo("amq.direct", response_queue));
```

Now send some requests...

```
string[] strs = {
                 "Twas brillig, and the slithy toves",
                 "Did gire and gymble in the wabe.",
                 "All mimsy were the borogroves,",
                 "And the mome raths outgrabe.",
                 "That's all, folks!"
                };
foreach (string s in strs)
{
 request.clearData();
 request.appendData(Encoding.UTF8.GetBytes(s));
 session.messageTransfer("amq.direct", request);
}
```

And wait for responses to arrive:

```
Console.WriteLine("Waiting for all responses to arrive ...");
Monitor.Wait(session);
```

1.1.1.7.3. The Server Application

The second program in the request-response example, Server.cs, uses the reply-to property as the routing key for responses.

The main body of Server.cs creates an exclusive queue for requests, then waits for messages to arrive.

```
const string request_queue = "request";
// Use the name of the request queue as the routing key
session.queueDeclare(request_queue);
session.exchangeBind(request_queue, "amq.direct", request_queue);

lock (session)
{
 // Create a listener and subscribe it to the request_queue
 IMessageListener listener = new MessageListener(session);
 session.attachMessageListener(listener, request_queue);
 session.messageSubscribe(request_queue);
 // Receive messages until all messages are received
 Console.WriteLine("Waiting for requests");
 Monitor.Wait(session);
}
```

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

```
BinaryReader reader = new BinaryReader(request.Body, Encoding.UTF8);
byte[] body = new byte[request.Body.Length - request.Body.Position];
reader.Read(body, 0, body.Length);
ASCIIEncoding enc = new ASCIIEncoding();
string message = enc.GetString(body);
Console.WriteLine("Request: " + message);

// Transform message content to upper case
string responseBody = message.ToUpper();

// Send it back to the user
response.clearData();
response.appendData(Encoding.UTF8.GetBytes(responseBody));
_session.messageTransfer("amq.direct", routingKey, response);
```

1.2. Excel AddIn

1.2.1. Excel AddIn

Qpid .net comes with Excel AddIns that are located in:

```
<project-root>\qpid\dotnet\client-010\addins
```

There are currently three projects:

ExcelAddIn	An RTD excel Addin
ExcelAddInProducer	A sample client to demonstrate the RTD AddIn
ExcelAddInMessageProcessor	A sample message processor for the RTD AddIn

1.2.1.1. Qpid RDT AddIn

1.2.1.1.1. Deploying the RTD AddIn

Excel provides a function called RTD (real-time data) that lets you specify a COM server via its ProgId here "Qpid" so that you can push qpid messages into Excel.

The provided RTD AddIn consumes messages from one queue and process them through a provided message processor.

For using the Qpid RTD follows those steps:

1. Copy the configuration Excel.exe.config into `Drive\Program Files\Microsoft Office \Office12`.
2. Edit `Excel.exe.xml` and set the targeted Qpid broker host, port number, username and password.
3. Select the cell or cell range to contain the RTD information
4. Enter the following formula **=rtd("Qpid",,"myQueue")**. Where MyQueue is the queue from which you wish to receive messages from.

Note: The Qpid RTD is a COM-AddIn that must be registered with Excel. This is done automatically when compiling the Addin with visual studio.

1.2.1.1.2. Defining a message processor

The default behavior of the RDT AddIn is to display the message payload. This could be altered by specifying your own message processor. A Message processor is a class that implements the API **ExcelAddIn.MessageProcessor**. For example, the provided processor in `client-010\addins \ExcelAddInMessageProcessor` displays the message body and the the header price when specified.

To use you own message processor follows those steps:

1. Write your own message processor that extends ExcelAddIn.MessageProcessor
2. Edit Excel.exe.config and uncomment the entries:

```
<add key="ProcessorAssembly"
value="<path>\qpid\dotnet\client-010\addins\ExcelAddInMessageProcessor\bin\Debu


            <add key="ProcessorClass"
            value="ExcelAddInMessageProcessor.Processor"/>
```

   - ProcessorAssembly is the path on the Assembly that contains your processor class
   - ProcessorClass is your processor class name

3. run excel and define a rtd function

Note: the provided ExcelAddInProducer can be used for testing the provided message processor. As messages are sent to queue1 the following rtd function should be used **=rtd("Qpid",,"queue1")**.

1.3. WCF

1.3.1. Introduction

WCF (*Windows Communication Foundation)* unifies the .Net communication capabilities into a single, common, general Web service oriented framework. A good WCF tutorial can be found here [http://www.netfxharmonics.com/2008/11/Understanding-WCF-Services-in-Silverlight-2#WCFSilverlightIntroduction].

WCF separates how service logic is written from how services communicate with clients. Bindings are used to specify the transport, encoding, and protocol details required for clients and services to communicate with each other. Qpid provide a WCF binding: org.apache.qpid.wcf.model.QpidBinding. WCF Services that use the Qpid binding communicate through queues that are dynamically created on a Qpid broker.

1.3.2. How to use Qpid binding

WCF services are implemented using:

- A service contract with one or more operation contracts.
- A service implementation for those contracts.
- A configuration file to provide that implementation with an endpoint and a binding for that specific contract.

The following configuration file can be used to configure a Hello Service:

```
<configuration>
  <system.serviceModel>
     <services>
      <!-- the service class -->
      <service name="org.apache.qpid.wcf.demo.HelloService">
        <host>
          <baseAddresses>
            <!-- Use SOAP over AMQP -->
            <add baseAddress="soap.amqp:///"   />
          </baseAddresses>
        </host>

        <endpoint
          address="Hello"
          <!-- We use a Qpid Binding, see below def -->
          binding="customBinding"
          bindingConfiguration="QpidBinding"
          <!-- The service contract -->
          contract="org.apache.qpid.wcf.demo.IHelloContract"/>
      </service>
    </services>

    <bindings>
      <customBinding>
        <!-- cf def of the qpid binding -->
        <binding name="QpidBinding">
```

```
          <textMessageEncoding />
          <!-- specify the host and port number of the broker -->
          <QpidTransport
               host="192.168.1.14"
               port="5673" />
        </binding>
      </customBinding>
    </bindings>

    <extensions>
      <bindingElementExtensions>
        <!-- use Qpid binding element: org.apache.qpid.wcf.model.QpidTransportElem
        <add
          name="QpidTransport"
           type="org.apache.qpid.wcf.model.QpidTransportElement, qpidWCFModel"/>
      </bindingElementExtensions>
    </extensions>

  </system.serviceModel>
</configuration>
```

Endpoints and bindings can also be set within the service code:

```
/* set HostName, portNumber and MyService accordingly */
Binding binding = new QpidBinding("HostName", portNumber);
ServiceHost service = new ServiceHost(typeof(MyService), new Uri("soap.amqp:///"))
service.AddServiceEndpoint(typeof(IBooking), binding, "MyService");
service.Open();
....
```

2. Examples

- http://svn.apache.org/viewvc/qpid/trunk/qpid/dotnet/client-010/examples/

Chapter 14. AMQP Python Messaging Client

1. User Guides

- Python Client API Guide [http://qpid.apache.org/docs/api/python/html/index.html]

2. Examples

- AMQP Python Client Examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/python/examples/]
- Running the AMQP Python Client Examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/python/examples/README]

3. PythonBrokerTest

3.1. Python Broker System Test Suite

This is a suite of python client tests that exercise and verify broker functionality. Python allows us to rapidly develop client test scenarios and provides a 'neutral' set of tests that can run against any AMQP-compliant broker.

The python/tests directory contains a collection of python modules, each containing several unittest classes, each containing a set of test methods that represent some test scenario. Test classes inherit qpid.TestBas from qpid/testlib.py, it inherits unittest.TestCase but adds some qpid-specific setUp/tearDown and convenience functions.

TODO: get pydoc generated up to qpid wiki or website automatically?

3.1.1. Running the tests

Simplest way to run the tests:

- Run a broker on the default port
- ./run_tests

For additional options: ./run_tests --help

3.1.2. Expected failures

Until we complete functionality, tests may fail because the tested functionality is missing in the broker. To skip expected failures in the C++ or Java brokers:

```
./run_tests -I cpp_failing.txt
./run_tests -I java_failing.txt
```

If you fix a failure, please remove it from the corresponding list.

Chapter 15. AMQP Ruby Messaging Client

The Ruby Messaging Client currently has little documentation and few examples.

1. Examples

AMQP Ruby Messaging Client Examples [https://svn.apache.org/repos/asf/qpid/trunk/qpid/ruby/examples]

Part V. Appendices

Table of Contents

Chapter 16. AMQP compatibility

Qpid provides the most complete and compatible implementation of AMQP. And is the most aggressive in implementing the latest version of the specification.

There are two brokers:

- C++ with support for AMQP 0-10
- Java with support for AMQP 0-8 and 0-9 (0-10 planned)

There are client libraries for C++, Java (JMS), .Net (written in C#), python and ruby.

- All clients support 0-10 and interoperate with the C++ broker.
- The JMS client supports 0-8, 0-9 and 0-10 and interoperates with both brokers.
- The python and ruby clients will also support all versions, but the API is dynamically driven by the specification used and so differs between versions. To work with the Java broker you must use 0-8 or 0-9, to work with the C++ broker you must use 0-10.
- There are two separate C# clients, one for 0-8 that interoperates with the Java broker, one for 0-10 that inteoperates with the C++ broker.

QMF Management is supported in Ruby, Python, C++, and via QMan for Java JMX & WS-DM.

1. AMQP Compatibility of Qpid releases:

Qpid implements the AMQP Specification, and as the specification has progressed Qpid is keeping up with the updates. This means that different Qpid versions support different versions of AMQP. Here is a simple guide on what use.

Here is a matrix that describes the different versions supported by each release. The status symbols are interpreted as follows:

Y supported

N unsupported

IP in progress

P planned

Table 16.1. AMQP Version Support by Qpid Release

Component	Spec				
		M2.1	M3	M4	0.5
java client	0-10		Y	Y	Y
	0-9	Y	Y	Y	Y
	0-8	Y	Y	Y	Y
java broker	0-10				P
	0-9	Y	Y	Y	Y

	0-8	Y	Y	Y	Y
c++ client/ broker	0-10		Y	Y	Y
	0-9	Y			
python client	0-10		Y	Y	Y
	0-9	Y	Y	Y	Y
	0-8	Y	Y	Y	Y
ruby client	0-10			Y	Y
	0-8	Y	Y	Y	Y
C# client	0-10			Y	Y
	0-8	Y	Y	Y	Y

2. Interop table by AMQP specification version

Above table represented in another format.

Table 16.2. AMQP Version Support - alternate format

	release	0-8	0-9	0-10
java client	M3 M4 0.5	Y	Y	Y
java client	M2.1	Y	Y	N
java broker	M3 M4 0.5	Y	Y	N
java broker	trunk	Y	Y	P
java broker	M2.1	Y	Y	N
c++ client/broker	M3 M4 0.5	N	N	Y
c++ client/broker	M2.1	N	Y	N
python client	M3 M4 0.5	Y	Y	Y
python client	M2.1	Y	Y	N
ruby client	M3 M4 0.5	Y	Y	N
ruby client	trunk	Y	Y	P
C# client	M3 M4 0.5	Y	N	N
C# client	trunk	Y	N	Y

Chapter 17. Qpid Interoperability Documentation

1. Qpid Interoperability Documentation

This page documents the various interoperable features of the Qpid clients.

1.1. SASL

1.1.1. Standard Mechanisms

http://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer#SASL_mechanisms

This table list the various SASL mechanisms that each component supports. The version listed shows when this functionality was added to the product.

Table 17.1. SASL Mechanism Support

Component	ANONYMOUS	CRAM-MD5	DIGEST-MD5	EXTERNAL	GSSAPI/ Kerberos	PLAIN
C++ Broker	M3[Section 1.1, “ Standard Mechanisms ” [177]] [1]	M3[Section 1.1, “ Standard Mechanisms ” [177]] [1], [2]			M3[Section 1.1, “ Standard Mechanisms ” [177]] [1], [2]	M1
C++ Client	M3[Section 1.1, “ Standard Mechanisms ” [177]] [1]					M1
Java Broker		M1				M1
Java Client		M1				M1
.Net Client	M2	M2	M2	M2		M2
Python Client						?
Ruby Client						?

1: Support for these will be in M3 (currently available on trunk).

2: C++ Broker uses Cyrus SASL [http://freshmeat.net/projects/cyrussasl/] which supports CRAM-MD5 and GSSAPI but these have not been tested yet

1.1.2. Custom Mechanisms

There have been some custom mechanisms added to our implementations.

Table 17.2. SASL Custom Mechanisms

Component	AMQPLAIN	CRAM-MD5-HASHED
C++ Broker		
C++ Client		
Java Broker	M1	M2
Java Client	M1	M2
.Net Client		
Python Client	M2	
Ruby Client	M2	

1.1.2.1. AMQPLAIN

1.1.2.2. CRAM-MD5-HASHED

The Java SASL implementations require that you have the password of the user to validate the incoming request. This then means that the user's password must be stored on disk. For this to be secure either the broker must encrypt the password file or the need for the password being stored must be removed.

The CRAM-MD5-HASHED SASL plugin removes the need for the plain text password to be stored on disk. The mechanism defers all functionality to the build in CRAM-MD5 module the only change is on the client side where it generates the hash of the password and uses that value as the password. This means that the Java Broker only need store the password hash on the file system. While a one way hash is not very secure compared to other forms of encryption in environments where the having the password in plain text is unacceptable this will provide and additional layer to protect the password. In particular this offers some protection where the same password may be shared amongst many systems. It offers no real extra protection against attacks on the broker (the secret is now the hash rather than the password).