# AMQP Messaging Broker (Implemented in C++)

# AMQP Messaging Broker (Implemented in C++)

# Table of Contents

Introduction ... vi
1. Running the AMQP Messaging Broker ... 1
  1.1. Running a Qpid C++ Broker ... 1
    1.1.1. Building the C++ Broker and Client Libraries ... 1
    1.1.2. Running the C++ Broker ... 1
    1.1.3. Most common questions getting qpidd running ... 1
    1.1.4. Authentication ... 2
    1.1.5. Slightly more complex configuration ... 3
    1.1.6. Loading extra modules ... 4
  1.2. Cheat Sheet for configuring Queue Options ... 5
    1.2.1. Configuring Queue Options ... 5
  1.3. Cheat Sheet for configuring Exchange Options ... 7
    1.3.1. Configuring Exchange Options ... 7
  1.4. Using Broker Federation ... 9
    1.4.1. Introduction ... 9
    1.4.2. What Is Broker Federation? ... 9
    1.4.3. The qpid-route Utility ... 9
    1.4.4. Example Scenarios ... 15
    1.4.5. Advanced Topics ... 16
  1.5. SSL ... 17
    1.5.1. SSL How to ... 17
  1.6. LVQ ... 18
    1.6.1. Understanding LVQ ... 18
    1.6.2. LVQ semantics: ... 19
    1.6.3. LVQ_NO_BROWSE semantics: ... 19
    1.6.4. LVQ Program Example ... 20
  1.7. Queue State Replication ... 24
    1.7.1. Asynchronous Replication of Queue State ... 24
  1.8. Starting a cluster ... 28
    1.8.1. Running a Qpidd cluster ... 28
  1.9. ACL ... 30
    1.9.1. v2 ACL file format for brokers ... 30
    1.9.2. Design Documentation ... 33
    1.9.3. v2 ACL User Guide ... 34
  1.10. AMQP compatibility ... 35
    1.10.1. AMQP Compatibility of Qpid releases: ... 36
    1.10.2. Interop table by AMQP specification version ... 37
  1.11. Qpid Interoperability Documentation ... 37
    1.11.1. Qpid Interoperability Documentation ... 37
2. Managing the AMQP Messaging Broker ... 40
  2.1. Managing the C++ Broker ... 40
    2.1.1. Using qpid-config ... 40
    2.1.2. Using qpid-route ... 42
    2.1.3. Using qpid-tool ... 43
    2.1.4. Using qpid-printevents ... 47
  2.2. Qpid Management Framework ... 47
    2.2.1. What Is QMF ... 47
    2.2.2. Getting Started with QMF ... 48
    2.2.3. QMF Concepts ... 48
    2.2.4. The QMF Protocol ... 52
    2.2.5. How to Write a QMF Console ... 52

2.2.6. How to Write a QMF Agent ... 52
2.3. QMF Python Console Tutorial ... 52
2.3.1. Prerequisite - Install Qpid Messaging ... 53
2.3.2. Synchronous Console Operations ... 53
2.3.3. Asynchronous Console Operations ... 58
2.3.4. Discovering what Kinds of Objects are Available ... 62

# List of Tables

1.1. Transport Options for Federation .... 14
1.2. ACL Support in Qpid Broker Versions .... 30
1.3. Mapping ACL Traps .... 33
1.4. Mapping Management Actions to ACL .... 34
1.5. AMQP Version Support by Qpid Release .... 36
1.6. AMQP Version Support - alternate format .... 37
1.7. SASL Mechanism Support .... 37
1.8. SASL Custom Mechanisms .... 38
2.1. XML Attributes for QMF Properties and Statistics .... 50
2.2. QMF Datatypes .... 51
2.3. XML Schema Mapping for QMF Types .... 51
2.4. QMF Python Console Class Methods .... 58

# Introduction

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 AMQP Compatibility to see which messaging clients work with each broker.

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

# Chapter 1. Running the AMQP Messaging Broker

## 1.1. Running a Qpid C++ Broker

### 1.1.1. 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
```

### 1.1.2. 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
```

### 1.1.3. Most common questions getting qpidd running

#### 1.1.3.1. 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

### 1.1.3.2. 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.

### 1.1.3.3. 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
```

### 1.1.3.4. 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.

## 1.1.4. Authentication

### 1.1.4.1. 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.

### 1.1.4.2. 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.

## 1.1.5. 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
```

### 1.1.6. 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)
```

## 1.2. Cheat Sheet for configuring Queue Options

### 1.2.1. Configuring Queue Options

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

- Section 1.2.1, “ Configuring Queue Options ”
- • Section 1.2.1.1, “ Applying Queue Sizing Constraints ”
  - Section 1.2.1.2, “ Changing the Queue ordering Behaviors (FIFO/LVQ) ”
  - Section 1.2.1.3, “ Setting additional behaviors ”
  - • Section 1.2.1.3.1, “ Persist Last Node ”
    - Section 1.2.1.3.2, “ Queue event generation ”
  - Section 1.2.1.4, “ Other Clients ”

#### 1.2.1.1. 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);
```

#### 1.2.1.2. 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 1.6.4, “ LVQ Program Example ” can be found here

#### 1.2.1.3. Setting additional behaviors

##### 1.2.1.3.1. 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);
```

#### 1.2.1.3.2. 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 1.7, " 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).

### 1.2.1.4. Other Clients

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

## 1.3. Cheat Sheet for configuring Exchange Options

### 1.3.1. 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

#### 1.3.1.1. 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");
```

#### 1.3.1.2. 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.

#### 1.3.1.3. 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);

...
```

## 1.4. Using Broker Federation

### 1.4.1. 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.

### 1.4.2. 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.

### 1.4.3. 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.

#### 1.4.3.1. 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.

#### 1.4.3.1.1. 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.

#### 1.4.3.1.2. 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.

### 1.4.3.2. 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.

### 1.4.3.3. 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.

#### 1.4.3.3.1. 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.

```
--------------------------------------------------+      +---------------------
                                 localhost:10002  |      |  localhost:10001
                                                  |      |
   +-------------+                                |      |      +--------------
   |             |                                |      |      |
   |             |  global.#    ---------------+  |      |      |
   |  amq.topic  |----------->   private queue |--------------->|  amq.topic
   |             |              ---------------+  |      |      |
   |             |                                |      |      |
   +-------------+                                |      |      +--------------
                                                  |      |
                                                  |      |
--------------------------------------------------+      +---------------------
```

## 1.4.3.3.2. 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
```

## 1.4.3.3.3. 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.

## 1.4.3.4. 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 1.1. 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.

## 1.4.3.5. Caveats, Limitations, and Things to Avoid

### 1.4.3.5.1. 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.

### 1.4.3.5.2. 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.

### 1.4.3.5.3. 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.

## 1.4.4. Example Scenarios

## 1.4.4.1. Using QPID to bridge disjoint IP networks

### 1.4.4.1.1. Multi-tiered topology

```
                        +-----+
                        |  5  |
                        +-----+
                      /         \
              +-----+             +-----+
              |  2  |             |  6  |
              +-----+             +-----+
            /    |    \              |    \
    +-----+   +-----+   +-----+   +-----+   +-----+
    |  1  |   |  3  |   |  4  |   |  7  |   |  8  |
    +-----+   +-----+   +-----+   +-----+   +-----+
```

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
```

##### 1.4.4.1.2. Load-sharing across brokers

### 1.4.5. Advanced Topics

#### 1.4.5.1. Federation Queue Naming

#### 1.4.5.2. Message Loop Prevention

## 1.5. SSL

### 1.5.1. SSL How to

#### 1.5.1.1. 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-
```

#### 1.5.1.2. 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
```

### 1.5.1.3. .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)

### 1.5.1.4. Python & Ruby Client (M4 and up)

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

## 1.6. LVQ

## 1.6.1. 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

## 1.6.2. 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
```

## 1.6.3. 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]$
```

## 1.6.4. 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;
    }
```

## 1.7. Queue State Replication

### 1.7.1. Asynchronous Replication of Queue State

#### 1.7.1.1. 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.

#### 1.7.1.2. 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.

#### 1.7.1.3. 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.

### 1.7.1.4. 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);
```

### 1.7.1.5. 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
```

## 1.8. Starting a cluster

### 1.8.1. Running a Qpidd cluster

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

#### 1.8.1.1. 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

### 1.8.1.2. 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

### 1.8.1.3. 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
}
```

### 1.8.1.4. 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.

### 1.8.1.5. 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.

#### 1.8.1.6. 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

## 1.9. ACL

### 1.9.1. 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 1.2. ACL Support in Qpid Broker Versions**

| Broker | Version |
|---|---|
| C++ | M4 onward |
| Java | M5 anticipated |

Contents

- Section 1.9.1, “ v2 ACL file format for brokers ”
- • Section 1.9.1.1, “ Specification ”
  - Section 1.9.1.2, “ Validation ”
  - Section 1.9.1.3, “ Example file: ”
- Section 1.9.2, “ Design Documentation ”
- • Section 1.9.2.1, “ Mapping of ACL traps to action and type ”
- Section 1.9.3, “ v2 ACL User Guide ”

- • Section 1.9.3.1, “ Writing Good/Fast ACL ”
  - Section 1.9.3.2, “ Getting ACL to Log ”
  - Section 1.9.3.3, “ User Id / domains running with C++ broker ”

### 1.9.1.1. 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
```

#### 1.9.1.2. 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.

#### 1.9.1.3. 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
```

## 1.9.2. Design Documentation

### 1.9.2.1. 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 1.3. Mapping ACL Traps**

| Object | Action | Properties | Trap C++ | Trap Java |
|---|---|---|---|---|
| Exchange | Create | name type alternate passive durable | ExchangeHandlerImpl::declare | 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::subscribe | 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 1.4. Mapping Management Actions to ACL**

| Object | Action | Property |
|---|---|---|
| Broker | Update | name=joinCluster |

## 1.9.3. v2 ACL User Guide

### 1.9.3.1. 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.

#### 1.9.3.2. 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
```

#### 1.9.3.3. 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
```

## 1.10. 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.

## 1.10.1. 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 1.5. 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 |

### 1.10.2. Interop table by AMQP specification version

Above table represented in another format.

**Table 1.6. 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 |

## 1.11. Qpid Interoperability Documentation

### 1.11.1. Qpid Interoperability Documentation

This page documents the various interoperable features of the Qpid clients.

#### 1.11.1.1. SASL

##### 1.11.1.1.1. 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 1.7. SASL Mechanism Support**

| Component | ANONYMOUS | CRAM-MD5 | DIGEST-MD5 | EXTERNAL | GSSAPI/ Kerberos | PLAIN |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| C++ Broker | M3 1 [Section 1.11.1.1.1, “ Standard Mechanisms ” [38]] [38], | M3 1 [Section 1.11.1.1.1, “ Standard Mechanisms ” Section 1.11.1.1.1, “ Standard Mechanisms ” [38]] | | [38], | M3 1 [Section 1.11.1.1.1, “ Standard Mechanisms ” Section 1.11.1.1.1, “ Standard Mechanisms ” [38]] | M1 |
| C++ Client | M3 1 [Section 1.11.1.1.1, “ Standard Mechanisms ” [38]] | | | | | 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

## 1.11.1.1.2. Custom Mechanisms

There have been some custom mechanisms added to our implementations.

### Table 1.8. 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 | |

#### 1.11.1.1.2.1. AMQPLAIN

#### 1.11.1.1.2.2. 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).

# Chapter 2. Managing the AMQP Messaging Broker

## 2.1. 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

### 2.1.1. 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
```

### 2.1.2. 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
```

### 2.1.3. 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:
```

### 2.1.4. 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!

## 2.2. Qpid Management Framework

- Section 2.2.1, “ What Is QMF ”
- Section 2.2.2, “ Getting Started with QMF ”
- Section 2.2.3, “ QMF Concepts ”
- • Section 2.2.3.1, “ Console, Agent, and Broker ”
  - Section 2.2.3.2, “ Schema ”
  - Section 2.2.3.3, “ Class Keys and Class Versioning ”
- Section 2.2.4, “ The QMF Protocol ”
- Section 2.2.5, “ How to Write a QMF Console ”
- Section 2.2.6, “ How to Write a QMF Agent ”

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

### 2.2.1. 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.

## 2.2.2. 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.

## 2.2.3. QMF Concepts

This section introduces important concepts underlying QMF.

### 2.2.3.1. 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.

```
   +-------------+    +---------+    +---------------+    +-------------------
   | CLI utility |    | Web app |    | Audit storage |    | Event correlation
   +-------------+    +---------+    +---------------+    +-------------------
          ^                ^                 ^                  ^          |
          |                |                 |                  |          |
          v                v                 v                  v          v
+--------------------------------------------------------------------------------
|                Qpid Messaging Bus (with QMF Broker capability)
+--------------------------------------------------------------------------------
                ^                     ^                     ^
                |                     |                     |
                v                     v                     v
       +----------------+    +----------------+    +----------------+
       | Manageable app |    | Manageable app |    | Manageable app |
       +----------------+    +----------------+    +----------------+
```

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.

### 2.2.3.2. 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.

#### 2.2.3.2.1. 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>
```

#### 2.2.3.2.2. 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>
```

#### 2.2.3.2.3. 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 2.1. 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 |

#### 2.2.3.2.4. 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>
```

#### 2.2.3.2.5. Event Classes

#### 2.2.3.2.6. 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 2.2. 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 2.3. 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.

#### 2.2.3.3. Class Keys and Class Versioning

### 2.2.4. 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 ???.

### 2.2.5. How to Write a QMF Console

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

### 2.2.6. How to Write a QMF Agent

## 2.3. QMF Python Console Tutorial

- Section 2.3.1, “ Prerequisite - Install Qpid Messaging ”

- Section 2.3.2, “ Synchronous Console Operations ”
- - Section 2.3.2.1, “ Creating a QMF Console Session and Attaching to a Broker ”
  - Section 2.3.2.2, “ Accessing Managed Objects ”
  - - Section 2.3.2.2.1, “ Viewing Properties and Statistics of an Object ”
    - Section 2.3.2.2.2, “ Invoking Methods on an Object ”
- Section 2.3.3, “ Asynchronous Console Operations ”
- - Section 2.3.3.1, “ Creating a Console Class to Receive Asynchronous Data ”
  - Section 2.3.3.2, “ Receiving Events ”
  - Section 2.3.3.3, “ Receiving Objects ”
  - Section 2.3.3.4, “ Asynchronous Method Calls and Method Timeouts ”
- Section 2.3.4, “ Discovering what Kinds of Objects are Available ”

## 2.3.1. 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.

## 2.3.2. 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.

### 2.3.2.1. 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.

#### 2.3.2.2. 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
>>>
```

#### 2.3.2.2.1. 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.

#### 2.3.2.2.2. 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.

### 2.3.3. 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.

#### 2.3.3.1. 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 2.4. 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.

### 2.3.3.2. 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
```

### 2.3.3.3. 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.

#### 2.3.3.4. 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.

### 2.3.4. Discovering what Kinds of Objects are Available