AMQP Messaging Broker (Implemented in C++)

AMQP Messaging Broker (Implemented in C++)

Table of Contents

List of Tables

List of Examples

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)
```

1.1.7. Timestamping Received Messages

The AMQP 0-10 specification defines a *timestamp* message delivery property. The timestamp delivery property is a *datetime* value that is written to each message that arrives at the broker. See the description of "message.delivery-properties" in the "Command Classes" section of the AMQP 0-10 specification for more detail.

See the *Programming in Apache Qpid* documentation for information regarding how clients may access the timestamp value in received messages.

By default, this timestamping feature is disabled. To enable timestamping, use the *enable-timestamp* broker configuration option. Setting the enable-timestamp option to 'yes' will enable message timestamping:

```
./qpidd --enable-timestamp yes
```

Message timestamping can also be enabled (and disabled) without restarting the broker. The QMF Broker management object defines two methods for accessing the timestamp configuration:

Table 1.1. QMF Management - Broker Methods for Managing the Timestamp Configuration

Method	Description
getTimestampConfig	Get the message timestamping configuration. Returns True if received messages are timestamped.
setTimestampConfig	Set the message timestamping configuration. Set True to enable timestamping received messages, False to disable timestamping.

Example 1.1. Enabling Message Timestamping via QMF - Python

The following code fragment uses these QMF method calls to enable message timestamping.

```
# get the state of the timestamp configuration
broker = self.qmf.getObjects(_class="broker")[0]
rc = broker.getTimestampConfig()
self.assertEqual(rc.status, 0)
self.assertEqual(rc.text, "OK")
print("The timestamp setting is %s" % str(rc.receive))

# try to enable it
rc = broker.setTimestampConfig(True)
self.assertEqual(rc.status, 0)
self.assertEqual(rc.text, "OK")
```

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.

The 0.10 C++ Broker supports the following additional Queue configuration options:

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 ??? 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. Broker Federation

Broker Federation allows messaging networks to be defined by creating *message routes*, in which messages in one broker (the *source broker*) are automatically routed to another broker (the *destination broker*). These routes may be defined between exchanges in the two brokers (the *source exchange* and the *destination exchange*), or from a queue in the source broker (the *source queue*) to an exchange in the destination broker. Message routes are unidirectional; when bidirectional flow is needed, one route is created in each direction. Routes can be durable or transient. A durable route survives broker restarts, restoring a route as soon as both the source broker and the destination are available. If the connection to a destination is lost, messages associated with a durable route continue to accumulate on the source, so they can be retrieved when the connection is reestablished.

Broker Federation can be used to build large messaging networks, with many brokers, one route at a time. If network connectivity permits, an entire distributed messaging network can be configured from a single location. The rules used for routing can be changed dynamically as servers change, responsibilities change, at different times of day, or to reflect other changing conditions.

Broker Federation is useful in a wide variety of scenarios. Some of these have to do with functional organization; for instance, brokers may be organized by geography, service type, or priority. Here are some use cases for federation:

- Geography: Customer requests may be routed to a processing location close to the customer.
- Service Type: High value customers may be routed to more responsive servers.
- Load balancing: Routing among brokers may be changed dynamically to account for changes in actual or anticipated load.
- High Availability: Routing may be changed to a new broker if an existing broker becomes unavailable.
- WAN Connectivity: Federated routes may connect disparate locations across a wide area network, while clients connect to brokers on their own local area network. Each broker can provide persistent queues that can hold messages even if there are gaps in WAN connectivity.
- Functional Organization: The flow of messages among software subsystems can be configured to mirror the logical structure of a distributed application.

- Replicated Exchanges: High-function exchanges like the XML exchange can be replicated to scale performance.

- Interdepartmental Workflow: The flow of messages among brokers can be configured to mirror interdepartmental workflow at an organization.

1.4.1. Message Routes

Broker Federation is done by creating message routes. The destination for a route is always an exchange on the destination broker. By default, a message route is created by configuring the destination broker, which then contacts the source broker to subscribe to the source queue. This is called a *pull route*. It is also possible to create a route by configuring the source broker, which then contacts the destination broker in order to send messages. This is called a *push route*, and is particularly useful when the destination broker may not be available at the time the messaging route is configured, or when a large number of routes are created with the same destination exchange.

The source for a route can be either an exchange or a queue on the source broker. If a route is between two exchanges, the routing criteria can be given explicitly, or the bindings of the destination exchange can be used to determine the routing criteria. To support this functionality, there are three kinds of message routes: queue routes, exchange routes, and dynamic exchange routes.

1.4.1.1. Queue Routes

Queue Routes route all messages from a source queue to a destination exchange. If message acknowledgement is enabled, messages are removed from the queue when they have been received by the destination exchange; if message acknowledgement is off, messages are removed from the queue when sent.

1.4.1.2. Exchange Routes

Exchange routes route messages from a source exchange to a destination exchange, using a binding key (which is optional for a fanout exchange).

Internally, creating an exchange route creates a private queue (auto-delete, exclusive) on the source broker to hold messages that are to be routed to the destination broker, binds this private queue to the source broker exchange, and subscribes the destination broker to the queue.

1.4.1.3. Dynamic Exchange Routes

Dynamic exchange routes allow a client to create bindings to an exchange on one broker, and receive messages that satisfy the conditions of these bindings not only from the exchange to which the client created the binding, but also from other exchanges that are connected to it using dynamic exchange routes. If the client modifies the bindings for a given exchange, they are also modified for dynamic exchange routes associated with that exchange.

Dynamic exchange routes apply all the bindings of a destination exchange to a source exchange, so that any message that would match one of these bindings is routed to the destination exchange. If bindings are added or removed from the destination exchange, these changes are reflected in the dynamic exchange route -- when the destination broker creates a binding with a given binding key, this is reflected in the route, and when the destination broker drops a binding with a binding key, the route no longer incurs the overhead of transferring messages that match the binding key among brokers. If two exchanges have dynamic exchange routes to each other, then all bindings in each exchange are reflected in the dynamic exchange route of the other. In a dynamic exchange route, the source and destination exchanges must have

the same exchange type, and they must have the same name; for instance, if the source exchange is a direct exchange, the destination exchange must also be a direct exchange, and the names must match.

Internally, dynamic exchange routes are implemented in the same way as exchange routes, except that the bindings used to implement dynamic exchange routes are modified if the bindings in the destination exchange change.

A dynamic exchange route is always a pull route. It can never be a push route.

1.4.2. Federation Topologies

A federated network is generally a tree, star, or line, using bidirectional links (implemented as a pair of unidirectional links) between any two brokers. A ring topology is also possible, if only unidirectional links are used.

Every message transfer takes time. For better performance, you should minimize the number of brokers between the message origin and final destination. In most cases, tree or star topologies do this best.

For any pair of nodes A,B in a federated network, there should be only one path from A to B. If there is more than one path, message loops can cause duplicate message transmission and flood the federated network. The topologies discussed above do not have message loops. A ring topology with bidirectional links is one example of a topology that does cause this problem, because a given broker can receive the same message from two different brokers. Mesh topologies can also cause this problem.

1.4.3. Federation among High Availability Message Clusters

Federation is generally used together with High Availability Message Clusters, using clusters to provide high availability on each LAN, and federation to route messages among the clusters. Because message state is replicated within a cluster, it makes little sense to define message routes between brokers in the same cluster.

To create a message route between two clusters, simply create a route between any one broker in the first cluster and any one broker in the second cluster. Each broker in a given cluster can use message routes defined for another broker in the same cluster. If the broker for which a message route is defined should fail, another broker in the same cluster can restore the message route.

1.4.4. The qpid-route Utility

qpid-route is a command line utility used to configure federated networks of brokers and to view the status and topology of networks. It can be used to configure routes among any brokers that **qpid-route** can connect to.

The syntax of **qpid-route** is as follows:

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

qpid-route [OPTIONS] route add <dest-broker> <src-broker> <exchange> <routing-key>
qpid-route [OPTIONS] route del <dest-broker> <src-broker> <exchange> <routing-key>

qpid-route [OPTIONS] queue add <dest-broker> <src-broker> <dest-exchange>  <src-qu
```

```
qpid-route [OPTIONS] queue del <dest-broker> <src-broker> <dest-exchange>  <src-qu

qpid-route [OPTIONS] list  [<broker>]
qpid-route [OPTIONS] flush [<broker>]
qpid-route [OPTIONS] map   [<broker>]


qpid-route [OPTIONS] list connections [<broker>]
```

The syntax for **broker**, **dest-broker**, and **src-broker** is as follows:

```
[username/password@] hostname | ip-address [:<port>]
```

The following are all valid examples of the above syntax: **localhost**, **10.1.1.7:10000**, **broker-host:10000**, **guest/guest@localhost**.

These are the options for **qpid-route**:

Table 1.2. qpid-route options

-v	Verbose output.
-q	Quiet output, will not print duplicate warnings.
-d	Make the route durable.
--timeout N	Maximum time to wait when qpid-route connects to a broker, in seconds. Default is 10 seconds.
--ack N	Acknowledge transfers of routed messages in batches of N. Default is 0 (no acknowledgements). Setting to 1 or greater enables acknowledgements; when using acknowledgements, values of N greater than 1 can significnantly improve performance, especially if there is significant network latency between the two brokers.
-s [--src-local]	Configure the route in the source broker (create a push route).
-t <transport> [--transport <transport>]	Transport protocol to be used for the route. • tcp (default) • ssl • rdma

1.4.4.1. Creating and Deleting Queue Routes

The syntax for creating and deleting queue routes is as follows:

```
qpid-route [OPTIONS] queue add <dest-broker> <src-broker> <dest-exchange> <src-que
qpid-route [OPTIONS] queue del <dest-broker> <src-broker> <dest-exchange> <src-que
```

For instance, the following creates a queue route that routes all messages from the queue named **public** on the source broker **localhost:10002** to the **amq.fanout** exchange on the destination broker **localhost:10001**:

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

If the **-d** option is specified, this queue route is persistent, and will be restored if one or both of the brokers is restarted:

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

The **del** command takes the same arguments as the **add** command. The following command deletes the queue route described above:

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

1.4.4.2. Creating and Deleting Exchange Routes

The syntax for creating and deleting exchange routes is as follows:

```
qpid-route [OPTIONS] route add <dest-broker> <src-broker> <exchange> <routing-key>
qpid-route [OPTIONS] route del <dest-broker> <src-broker> <exchange> <routing-key>
qpid-route [OPTIONS] flush [<broker>]
```

For instance, the following creates an exchange route that routes messages that match the binding key **global.#** from the **amq.topic** exchange on the source broker **localhost:10002** to the **amq.topic** exchange on the destination broker **localhost:10001**:

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

In many applications, messages published to the destination exchange should also be routed to the source exchange. This is accomplished by creating a second exchange route, reversing the roles of the two exchanges:

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

If the **-d** option is specified, the exchange route is persistent, and will be restored if one or both of the brokers is restarted:

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

The **del** command takes the same arguments as the **add** command. The following command deletes the first exchange route described above:

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

1.4.4.3. Deleting all routes for a broker

Use the **flush** command to delete all routes for a given broker:

```
qpid-route [OPTIONS] flush [<broker>]
```

For instance, the following command deletes all routes for the broker **localhost:10001**:

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

1.4.4.4. Creating and Deleting Dynamic Exchange Routes

The syntax for creating and deleting dynamic exchange routes is as follows:

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

In the following examples, we will route messages from a topic exchange. We will create a new topic exchange and federate it so that we are not affected by other all clients that use the built-in **amq.topic** exchange. The following commands create a new topic exchange on each of two brokers:

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

Now let's create a dynamic exchange route that routes messages from the **fed.topic** exchange on the source broker **localhost:10004** to the **fed.topic** exchange on the destination broker **localhost:10003** if they match any binding on the destination broker's **fed.topic** exchange:

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

Internally, this creates a private autodelete queue on the source broker, and binds that queue to the **fed.topic** exchange on the source broker, using each binding associated with the **fed.topic** exchange on the destination broker.

In many applications, messages published to the destination exchange should also be routed to the source exchange. This is accomplished by creating a second dynamic exchange route, reversing the roles of the two exchanges:

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

If the **-d** option is specified, the exchange route is persistent, and will be restored if one or both of the brokers is restarted:

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

When an exchange route is durable, the private queue used to store messages for the route on the source exchange is also durable. If the connection between the brokers is lost, messages for the destination exchange continue to accumulate until it can be restored.

The **del** command takes the same arguments as the **add** command. The following command deletes the first exchange route described above:

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

Internally, this deletes the bindings on the source exchange for the the private queues associated with the message route.

1.4.4.5. Viewing Routes

The **route list** command shows the routes associated with an individual broker. For instance, suppose we have created the following two routes:

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

We can now use **route list** to show all routes for the broker **localhost:10003**:

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

Note that this shows only one of the two routes we created, the route for which **localhost:10003** is a destination. If we want to see the route for which **localhost:10004** is a destination, we need to do another route list:

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

The **route map** command shows all routes associated with a broker, and recursively displays all routes for brokers involved in federation relationships with the given broker. For instance, here is the output for the two brokers configured above:

```
$ 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
```

Note that the two dynamic exchange links are displayed as though they were one bidirectional link. The **route map** command is particularly helpful for larger, more complex networks. Let's configure a somewhat more complex network with 16 dynamic exchange routes:

```
qpid-route dynamic add localhost:10001 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10001 fed.topic

qpid-route dynamic add localhost:10003 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10003 fed.topic
```

```
qpid-route dynamic add localhost:10004 localhost:10002 fed.topic
qpid-route dynamic add localhost:10002 localhost:10004 fed.topic

qpid-route dynamic add localhost:10002 localhost:10005 fed.topic
qpid-route dynamic add localhost:10005 localhost:10002 fed.topic

qpid-route dynamic add localhost:10005 localhost:10006 fed.topic
qpid-route dynamic add localhost:10006 localhost:10005 fed.topic

qpid-route dynamic add localhost:10006 localhost:10007 fed.topic
qpid-route dynamic add localhost:10007 localhost:10006 fed.topic

qpid-route dynamic add localhost:10006 localhost:10008 fed.topic
qpid-route dynamic add localhost:10008 localhost:10006 fed.topic
```

Now we can use **route map** starting with any one broker, and see the entire network:

```
$ ./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

Static Routes:
  none found
```

1.4.4.6. Resilient Connections

When a broker route is created, or when a durable broker route is restored after broker restart, a connection is created between the source broker and the destination broker. The connections used between brokers are called *resilient connections*; if the connection fails due to a communication error, it attempts to reconnect. The retry interval begins at 2 seconds and, as more attempts are made, grows to 64 seconds, and continues to retry every 64 seconds thereafter. If the connection fails due to an authentication problem, it will not continue to retry.

The command **list connections** can be used to show the resilient connections for a broker:

```
$ qpid-route list connections 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
```

In the above output, **Last Error** contains the string representation of the last connection error received for the connection. **State** represents the state of the connection, and may be one of the following values:

Table 1.3. State values in $ qpid-route list connections

Waiting	Waiting before attempting to reconnect.
Connecting	Attempting to establish the connection.
Operational	The connection has been established and can be used.
Failed	The connection failed and will not retry (usually because authentication failed).
Closed	The connection has been closed and will soon be deleted.
Passive	If a cluster is federated to another cluster, only one of the nodes has an actual connection to remote node. Other nodes in the cluster have a passive connection.

1.5. Security

This chapter describes how authentication, rule-based authorization, encryption, and digital signing can be accomplished using Qpid. Authentication is the process of verifying the identity of a user; in Qpid, this is done using the SASL framework. Rule-based authorization is a mechanism for specifying the actions that each user is allowed to perform; in Qpid, this is done using an Access Control List (ACL) that is part of the Qpid broker. Encryption is used to ensure that data is not transferred in a plain-text format that could be intercepted and read. Digital signatures provide proof that a given message was sent by a known sender. Encryption and signing are done using SSL (they can also be done using SASL, but SSL provides stronger encryption).

1.5.1. User Authentication

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

Both the Qpid broker and Qpid clients use the Cyrus SASL library [http://cyrusimap.web.cmu.edu/], a full-featured authentication framework, which offers many configuration options. This section shows how to configure users for authentication with SASL, which is sufficient when using **SASL PLAIN**. If you are not using SSL, you should configure SASL to use **CRAM-MD5**, **DIGEST-MD5**, or **GSSAPI** (which provides Kerberos authentication). For information on configuring these and other options in SASL, see the Cyrus SASL documentation.

Important

The **SASL PLAIN** method sends passwords in cleartext, and is vulnerable to man-in-the-middle attacks unless SSL (Secure Socket Layer) is also used (see Section 1.5.3, “Encryption using SSL”).

If you are not using SSL, we recommend that you disable **PLAIN** authentication in the broker.

The Qpid broker uses the **auth yes|no** option to determine whether to use SASL authentication. Turn on authentication by setting **auth** to **yes** in `/etc/qpidd.conf`:

```
# /etc/qpidd.conf
#
# Set auth to 'yes' or 'no'

auth=yes
```

1.5.1.1. Configuring SASL

On Linux systems, the SASL configuration file is generally found in `/etc/sasl2/qpidd.conf` or `/usr/lib/sasl2/qpidd.conf`.

The SASL database contains user names and passwords for SASL. In SASL, a user may be associated with a *realm*. The Qpid broker authenticates users in the **QPID** realm by default, but it can be set to a different realm using the **realm** option:

```
# /etc/qpidd.conf
#
# Set the SASL realm using 'realm='

auth=yes
realm=QPID
```

The SASL database is installed at `/var/lib/qpidd/qpidd.sasldb`; initially, it has one user named **guest** in the **QPID** realm, and the password for this user is **guest**.

Note

The user database is readable only by the `qpidd` user. When run as a daemon, Qpid always runs as the `qpidd` user. If you start the broker from a user other than the `qpidd` user, you will need to either reconfigure SASL or turn authentication off.

Important

The SASL database stores user names and passwords in plain text. If it is compromised so are all of the passwords that it stores. This is the reason that the `qpidd` user is the only user that can read the database. If you modify permissions, be careful not to expose the SASL database.

Add new users to the database by using the **saslpasswd2** command, which specifies a realm and a user ID. A user ID takes the form ***user-id@domain***..

```
# saslpasswd2 -f /var/lib/qpidd/qpidd.sasldb -u realm new_user_name
```

To list the users in the SASL database, use **sasldblistusers2**:

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

If you are using **PLAIN** authentication, users who are in the database can now connect with their user name and password. This is secure only if you are using SSL. If you are using a more secure form of authentication, please consult your SASL documentation for information on configuring the options you need.

1.5.1.2. Kerberos

Both the Qpid broker and Qpid users are 'principals' of the Kerberos server, which means that they are both clients of the Kerberos authentication services.

To use Kerberos, both the Qpid broker and each Qpid user must be authenticated on the Kerberos server:

1. Install the Kerberos workstation software and Cyrus SASL GSSAPI on each machine that runs a qpidd broker or a qpidd messaging client:

   ```
   $ sudo yum install cyrus-sasl-gssapi krb5-workstation
   ```

2. Make sure that the Qpid broker is registered in the Kerberos database.

   Traditionally, a Kerberos principal is divided into three parts: the primary, the instance, and the realm. A typical Kerberos V5 has the format `primary/instance@REALM`. For a Qpid broker, the primary is `qpidd`, the instance is the fully qualified domain name, which you can obtain using **hostname --fqdn**, and the REALM is the Kerberos domain realm. By default, this realm is `QPID`, but a different realm can be specified in qpid.conf, e.g.:

   ```
   realm=EXAMPLE.COM
   ```

   For instance, if the fully qualified domain name is `dublduck.example.com` and the Kerberos domain realm is `EXAMPLE.COM`, then the principal name is `qpidd/dublduck.example.com@EXAMPLE.COM`.

   The following script creates a principal for qpidd:

   ```
   FDQN=`hostname --fqdn`
   REALM="EXAMPLE.COM"
   kadmin -r $REALM  -q "addprinc -randkey -clearpolicy qpidd/$FQDN"
   ```

   Now create a Kerberos keytab file for the Qpid broker. The Qpid broker must have read access to the keytab file. The following script creates a keytab file and allows the broker read access:

   ```
   QPIDD_GROUP="qpidd"
   kadmin -r $REALM  -q "ktadd -k /etc/qpidd.keytab qpidd/$FQDN@$REALM"
   chmod g+r /etc/qpidd.keytab
   chgrp $QPIDD_GROUP /etc/qpidd.keytab
   ```

   The default location for the keytab file is `/etc/krb5.keytab`. If a different keytab file is used, the KRB5_KTNAME environment variable must contain the name of the file, e.g.:

   ```
   export KRB5_KTNAME=/etc/qpidd.keytab
   ```

If this is correctly configured, you can now enable kerberos support on the Qpid broker by setting the `auth` and `realm` options in `/etc/qpidd.conf`:

```
# /etc/qpidd.conf
auth=yes
realm=EXAMPLE.COM
```

Restart the broker to activate these settings.

3. Make sure that each Qpid user is registered in the Kerberos database, and that Kerberos is correctly configured on the client machine. The Qpid user is the account from which a Qpid messaging client is run. If it is correctly configured, the following command should succeed:

   ```
   $ kinit user@REALM.COM
   ```

Java JMS clients require a few additional steps.

1. The Java JVM must be run with the following arguments:

-Djavax.security.auth.useSubjectCredsOnly=false	Forces the SASL GASSPI client to obtain the kerberos credentials explicitly instead of obtaining from the "subject" that owns the current thread.

-Djava.security.auth.login.config=myjas.conf	Specifies the jass configuration file. Here is a sample JASS configuration file:
	`com.sun.security.jgss.initiate {`
	`com.sun.security.auth.module.Krb5Lo`
	`};`

-Dsun.security.krb5.debug=true	Enables detailed debug info for troubleshooting

2. The client's Connection URL must specify the following Kerberos-specific broker properties:

   - `sasl_mechs` must be set to `GSSAPI`.
   - `sasl_protocol` must be set to the principal for the qpidd broker, e.g. `qpidd`/
   - `sasl_server` must be set to the host for the SASL server, e.g. `sasl.com`.

   Here is a sample connection URL for a Kerberos connection:

   ```
   amqp://guest@clientid/testpath?brokerlist='tcp://localhost:5672?sasl_mechs='GSS
   ```

1.5.2. Authorization

In Qpid, Authorization specifies which actions can be performed by each authenticated user using an Access Control List (ACL). Use the **--acl-file** command to load the access control list. The filename should have a `.acl` extension:

```
$ qpidd --acl-file ./aclfilename.acl
```

Each line in an ACL file grants or denies specific rights to a user. If the last line in an ACL file is `acl deny all all`, the ACL uses *deny mode*, and only those rights that are explicitly allowed are granted:

```
acl allow rajith@QPID all all
acl deny all all
```

On this server, `rajith@QPID` can perform any action, but nobody else can. Deny mode is the default, so the previous example is equivalent to the following ACL file:

```
acl allow rajith@QPID all all
```

In deny mode, denying rights to an action is redundant and has no effect.

```
acl allow rajith@QPID all all
acl deny jonathan@QPID all all # This rule is redundant, and has no effect
acl deny all all
```

If the last line in an ACL file is `acl allow all all`, ACL uses *allow mode*, and all rights are granted except those that are explicitly denied. The following ACL file allows everyone else to perform any action, but denies `jonathan@QPID` all permissions.

```
acl deny jonathan@QPID all all
acl allow all all
```

In allow mode, allowing rights to an action is redundant and has no effect.

```
acl allow rajith@QPID all all # This rule is redundant, and has no effect
acl deny jonathan@QPID all all
acl allow all all
```

Important

ACL processing ends when one of the following lines is encountered:

```
acl allow all all
```

```
acl deny all all
```

Any lines that occur after one of these statements will be ignored:

```
acl allow all all
acl deny jonathan@QPID all all # This line is ignored !!!
```

ACL syntax allows fine-grained access rights for specific actions:

```
acl allow carlt@QPID create exchange name=carl.*
acl allow fred@QPID create all
acl allow all consume queue
acl allow all bind exchange
acl deny all all
```

An ACL file can define user groups, and assign permissions to them:

```
group admin ted@QPID martin@QPID
acl allow admin create all
acl deny all all
```

1.5.2.1. ACL Syntax

ACL rules must be on a single line and follow this syntax:

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

ACL rules can also include a single object name (or the keyword *all*) and one or more property name value pairs in the form **property=value**

The following tables show the possible values for **permission**, **action**, **object**, and **property** in an ACL rules file.

Table 1.4. ACL Rules: permission

allow	Allow the action
allow-log	Allow the action and log the action in the event log
deny	Deny the action
deny-log	Deny the action and log the action in the event log

Table 1.5. ACL Rules:action

consume	Applied when subscriptions are created
publish	Applied on a per message basis on publish message transfers, this rule consumes the most resources
create	Applied when an object is created, such as bindings, queues, exchanges, links
access	Applied when an object is read or accessed
bind	Applied when objects are bound together
unbind	Applied when objects are unbound
delete	Applied when objects are deleted
purge	Similar to delete but the action is performed on more than one object
update	Applied when an object is updated

Table 1.6. ACL Rules:object

queue	A queue

exchange	An exchange
broker	The broker
link	A federation or inter-broker link
method	Management or agent or broker method

Table 1.7. ACL Rules:property

name	String. Object name, such as a queue name or exchange name.
durable	Boolean. Indicates the object is durable
routingkey	Sring. Specifies routing key
passive	Boolean. Indicates the presence of a `passive` flag
autodelete	Boolean. Indicates whether or not the object gets deleted when the connection is closed
exclusive	Boolean. Indicates the presence of an `exclusive` flag
type	String. Type of object, such as topic, fanout, or xml
alternate	String. Name of the alternate exchange
queuename	String. Name of the queue (used only when the object is something other than `queue`
schemapackage	String. QMF schema package name
schemaclass	String. QMF schema class name

1.5.2.2. ACL Syntactic Conventions

In ACL files, the following syntactic conventions apply:

- A line starting with the # character is considered a comment and is ignored.
- Empty lines and lines that contain only whitespace are ignored
- All tokens are case sensitive. `name1` is not the same as `Name1` and `create` is not the same as `CREATE`
- Group lists can be extended to the following line by terminating the line with the \ character
- Additional whitespace - that is, where there is more than one whitespace character - between and after tokens is ignored. Group and ACL definitions must start with either **group** or **acl** 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` matches all individuals, groups and actions
- The last line of the file - whether present or not - will be assumed to be **acl deny all all**. If present in the file, all lines below it are ignored.

- Names and group names may contain only *a-z*, *A-Z*, *0-9*, - and _
- Rules must be preceded by any group definitions they can use. Any name not defined as a group will be assumed to be that of an individual.

1.5.2.3. Specifying ACL Permissions

Now that we have seen the ACL syntax, we will provide representative examples and guidelines for ACL files.

Most ACL files begin by defining 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
```

Rules in an ACL file grant or deny specific permissions to users or groups:

```
acl allow carlt@QPID create exchange name=carl.*
acl allow rob@QPID create queue
acl allow guest@QPID bind exchange name=amq.topic routingkey=stocks.rht.#
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 allow all consume queue
acl allow all bind exchange
acl deny all all
```

In the previous example, the last line, `acl deny all all`, denies all authorizations that have not been specifically granted. This is the default, but it is useful to include it explicitly on the last line for the sake of clarity. If you want to grant all rights by default, you can specify `acl allow all all` in the last line.

Do not allow *guest* to access and log QMF management methods that could cause security breaches:

```
group allUsers guest@QPID
....
acl deny-log allUsers create link
acl deny-log allUsers access method name=connect
acl deny-log allUsers access method name=echo
acl allow all all
```

1.5.3. Encryption using SSL

Encryption and certificate management for **qpidd** is provided by Mozilla's Network Security Services Library (NSS).

Enabling SSL for the Qpid broker

1. You will need a certificate that has been signed by a Certification Authority (CA). This certificate will also need to be trusted by your client. If you require client authentication in addition to server authentication, the client's certificate will also need to be signed by a CA and trusted by the broker.

   In the broker, SSL is provided through the **ssl.so** module. This module is installed and loaded by default in Qpid. To enable the module, you need to specify the location of the database containing the certificate and key to use. This is done using the **ssl-cert-db** option.

   The certificate database is created and managed by the Mozilla Network Security Services (NSS) **certutil** tool. Information on this utility can be found on the Mozilla website [http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html], including tutorials on setting up and testing SSL connections. The certificate database will generally be password protected. The safest way to specify the password is to place it in a protected file, use the password file when creating the database, and specify the password file with the **ssl-cert-password-file** option when starting the broker.

   The following script shows how to create a certificate database using certutil:

   ```
   mkdir ${CERT_DIR}
   certutil -N -d ${CERT_DIR} -f ${CERT_PW_FILE}
   certutil -S -d ${CERT_DIR} -n ${NICKNAME} -s "CN=${NICKNAME}" -t "CT,," -x -f ${
   ```

   When starting the broker, set **ssl-cert-password-file** to the value of **${CERT_PW_FILE}**, set **ssl-cert-db** to the value of **${CERT_DIR}**, and set **ssl-cert-name** to the value of **${NICKNAME}**.

2. The following SSL options can be used when starting the broker:

--ssl-use-export-policy	Use NSS export policy
--ssl-cert-password-file *PATH*	Required. Plain-text file containing password to use for accessing certificate database.
--ssl-cert-db *PATH*	Required. Path to directory containing certificate database.
--ssl-cert-name *NAME*	Name of the certificate to use. Default is `localhost.localdomain`.
--ssl-port *NUMBER*	Port on which to listen for SSL connections. If no port is specified, port 5671 is used.
--ssl-require-client-authentication	Require SSL client authentication (i.e. verification of a client certificate) during the SSL handshake. This occurs before SASL authentication, and is independent of SASL. This option enables the `EXTERNAL` SASL mechanism for SSL connections. If the client chooses the `EXTERNAL` mechanism, the client's identity is taken from the validated SSL certificate,

using the CNliteral>, and appending any DCliteral>s to create the domain. For instance, if the certificate contains the properties CN=bob, DC=acme, DC=com, the client's identity is bob@acme.com.

If the client chooses a different SASL mechanism, the identity take from the client certificate will be replaced by that negotiated during the SASL handshake.

--ssl-sasl-no-dict Do not accept SASL mechanisms that can be compromised by dictionary attacks. This prevents a weaker mechanism being selected instead of EXTERNAL, which is not vulnerable to dictionary attacks.

Also relevant is the **--require-encryption** broker option. This will cause **qpidd** to only accept encrypted connections.

Enabling SSL in Clients

C++ clients:

1. In C++ clients, SSL is implemented in the **sslconnector.so** module. This module is installed and loaded by default in Qpid.

   The following options can be specified for C++ clients using environment variables:

Table 1.8. SSL Client Environment Variables for C++ clients

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

2. When using SSL connections, clients must specify the location of the certificate database, a directory that contains the client's certificate and the public key of the Certificate Authority. This can be done by setting the environment variable **QPID_SSL_CERT_DB** to the full pathname of the directory. If a connection uses SSL client authentication, the client's password is also needed —the password should be placed in a protected file, and the **QPID_SSL_CERT_PASSWORD_FILE** variable should be set to the location of the file containing this password.

3. To open an SSL enabled connection in the Qpid Messaging API, set the *protocol* connection option to *ssl*.

Java clients:

1. For both server and client authentication, import the trusted CA to your trust store and keystore and generate keys for them. Create a certificate request using the generated keys and then create a certificate using the request. You can then import the signed certificate into your keystore. Pass the following arguments to the Java JVM when starting 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
```

2. For server side authentication only, import the trusted CA to your trust store and pass the following arguments to the Java JVM when starting your client:

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

3. Java clients must use the SSL option in the connection URL to enable SSL encryption, e.g.

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

4. If you need to debug problems in an SSL connection, enable Java's SSL debugging by passing the argument `-Djavax.net.debug=ssl` to the Java JVM when starting your client.

1.6. LVQ - Last Value Queue

1.6.1. Understanding LVQ

A Last Value Queue is configured with the name of a message header that is used as a key. The queue behaves as a normal FIFO queue with the exception that when a message is enqueued, any other message in the queue with the same value in the key header is removed and discarded. Thus, for any given key value, the queue holds only the most recent message.

The following example illustrates the operation of a Last Value Queue. The example shows an empty queue with no consumers and a sequence of produced messages. The numbers represent the key for each message.

```
     <empty queue>
1 =>
     1
2 =>
     1 2
3 =>
     1 2 3
4 =>
     1 2 3 4
2 =>
```

```
     1 3 4 2
1 =>
     3 4 2 1
```

Note that the first four messages are enqueued normally in FIFO order. The fifth message has key '2' and is also enqueued on the tail of the queue. However the message already in the queue with the same key is discarded.

Note

If the set of keys used in the messages in a LVQ is constrained, the number of messages in the queue shall not exceed the number of distinct keys in use.

1.6.1.1. Common Use-Cases

- LVQ with zero or one consuming subscriptions - In this case, if the consumer drops momentarily or is slower than the producer(s), it will only receive current information relative to the message keys.
- LVQ with zero or more browsing subscriptions - A browsing consumer can subscribe to the LVQ and get an immediate dump of all of the "current" messages and track updates thereafter. Any number of independent browsers can subscribe to the same LVQ with the same effect. Since messages are never consumed, they only disappear when replaced with a newer message with the same key or when their TTL expires.

1.6.2. Creating a Last Value Queue

1.6.2.1. Using Addressing Syntax

A LVQ may be created using directives in the API's address syntax. The important argument is "qpid.last_value_queue_key". The following Python example shows how a producer of stock price updates can create a LVQ to hold the latest stock prices for each ticker symbol. The message header used to hold the ticker symbol is called "ticker".

```
conn = Connection(url)
conn.open()
sess = conn.session()
tx = sess.sender("prices;{create:always, node:{type:queue, x-declare:{argument
```

1.6.2.2. Using qpid-config

The same LVQ as shown in the previous example can be created using the qpid-config utility:

```
$ qpid-config add queue prices --lvq-key ticker
```

1.6.3. LVQ Example

1.6.3.1. LVQ Sender

```
from qpid.messaging import Connection, Message

def send(sender, key, message):
  message.properties["ticker"] = key
  sender.send(message)

conn = Connection("localhost")
conn.open()
sess = conn.session()
tx = sess.sender("prices;{create:always, node:{type:queue,x-declare:{arguments

msg = Message("Content")
send(tx, "key1", msg);
send(tx, "key2", msg);
send(tx, "key3", msg);
send(tx, "key4", msg);
send(tx, "key2", msg);
send(tx, "key1", msg);

conn.close()
```

1.6.3.2. LVQ Browsing Receiver

```
from qpid.messaging import Connection, Message

conn = Connection("localhost")
conn.open()
sess = conn.session()
rx = sess.receiver("prices;{mode:browse}")

while True:
  msg = rx.fetch()
  sess.acknowledge()
  print msg
```

1.6.4. Deprecated LVQ Modes

There are two legacy modes (still implemented as of Qpid 0.14) controlled by the qpid.last_value_queue and qpid.last_value_queue_no_browse argument values. These modes are deprecated and should not be used.

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. Active-active Messaging Clusters

Active-active Messaging Clusters provide fault tolerance by ensuring that every broker in a *cluster* has the same queues, exchanges, messages, and bindings, and allowing a client to *fail over* to a new broker and continue without any loss of messages if the current broker fails or becomes unavailable. *Active-active* refers to the fact that all brokers in the cluster can actively serve clients. Because all brokers are automatically kept in a consistent state, clients can connect to and use any broker in a cluster. Any number of messaging brokers can be run as one *cluster*, and brokers can be added to or removed from a cluster while it is in use.

High Availability Messaging Clusters are implemented using using the OpenAIS Cluster Framework [http://www.openais.org/].

An OpenAIS daemon runs on every machine in the cluster, and these daemons communicate using multicast on a particular address. Every qpidd process in a cluster joins a named group that is automatically synchronized using OpenAIS Closed Process Groups (CPG) — the qpidd processes multicast events to the named group, and CPG ensures that each qpidd process receives all the events in the same sequence. All members get an identical sequence of events, so they can all update their state consistently.

Two messaging brokers are in the same cluster if

1. They run on hosts in the same OpenAIS cluster; that is, OpenAIS is configured with the same mcastaddr, mcastport and bindnetaddr, and

2. They use the same cluster name.

High Availability Clustering has a cost: in order to allow each broker in a cluster to continue the work of any other broker, a cluster must replicate state for all brokers in the cluster. Because of this, the brokers in a cluster should normally be on a LAN; there should be fast and reliable connections between brokers. Even on a LAN, using multiple brokers in a cluster is somewhat slower than using a single broker without clustering. This may be counter-intuitive for people who are used to clustering in the context of High Performance Computing or High Throughput Computing, where clustering increases performance or throughput.

High Availability Messaging Clusters should be used together with Red Hat Clustering Services (RHCS); without RHCS, clusters are vulnerable to the "split-brain" condition, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. See the documentation on the **--cluster-cman** option for details on running using RHCS with High Availability Messaging Clusters. See the CMAN Wiki [http://sources.redhat.com/cluster/wiki] for more detail on CMAN and split-brain conditions. Use the **--cluster-cman** option to enable RHCS when starting the broker.

1.8.1. Starting a Broker in a Cluster

Clustering is implemented using the `cluster.so` module, which is loaded by default when you start a broker. To run brokers in a cluster, make sure they all use the same OpenAIS mcastaddr, mcastport, and bindnetaddr. All brokers in a cluster must also have the same cluster name — specify the cluster name in `qpidd.conf`:

```
cluster-name="local_test_cluster"
```

On RHEL6, you must create the file `/etc/corosync/uidgid.d/qpidd` to tell Corosync the name of the user running the broker.By default, the user is qpidd:

```
uidgid {
uid: qpidd
gid: qpidd
}
```

On RHEL5, the primary group for the process running qpidd must be the ais group. If you are running qpidd as a service, it is run as the **qpidd** user, which is already in the ais group. If you are running the broker from the command line, you must ensure that the primary group for the user running qpidd is ais. You can set the primary group using **newgrp**:

```
$ newgrp ais
```

You can then run the broker from the command line, specifying the cluster name as an option.

```
[jonathan@localhost]$ qpidd --cluster-name="local_test_cluster"
```

All brokers in a cluster must have identical configuration, with a few exceptions noted below. They must load the same set of plug-ins, and have matching configuration files and command line arguments. The should also have identical ACL files and SASL databases if these are used. If one broker uses persistence, all must use persistence — a mix of transient and persistent brokers is not allowed. Differences in configuration can cause brokers to exit the cluster. For instance, if different ACL settings allow a client to access a queue on broker A but not on broker B, then publishing to the queue will succeed on A and fail on B, so B will exit the cluster to prevent inconsistency.

The following settings can differ for brokers on a given cluster:

- logging options
- cluster-url — if set, it will be different for each broker.
- port — brokers can listen on different ports.

The qpid log contains entries that record significant clustering events, e.g. when a broker becomes a member of a cluster, the membership of a cluster is changed, or an old journal is moved out of the way. For instance, the following message states that a broker has been added to a cluster as the first node:

```
2009-07-09 18:13:41 info 127.0.0.1:1410(READY) member update: 127.0.0.1:1410
2009-07-09 18:13:41 notice 127.0.0.1:1410(READY) first in cluster
```

Note

If you are using SELinux, the qpidd process and OpenAIS must have the same SELinux context, or else SELinux must be set to permissive mode. If both qpidd and OpenAIS are run as services,

they have the same SELinux context. If both OpenAIS and qpidd are run as user processes, they have the same SELinux context. If one is run as a service, and the other is run as a user process, they have different SELinux contexts.

The following options are available for clustering:

Table 1.9. Options for High Availability Messaging Cluster

Options for High Availability Messaging Cluster	
--cluster-name *NAME*	Name of the Messaging Cluster to join. A Messaging Cluster consists of all brokers started with the same cluster-name and openais configuration.
--cluster-size *N*	Wait for at least N initial members before completing cluster initialization and serving clients. Use this option in a persistent cluster so all brokers in a persistent cluster can exchange the status of their persistent store and do consistency checks before serving clients.

--cluster-url *URL*	An AMQP URL containing the local address that the broker advertizes to clients for fail-over connections. This is different for each host. By default, all local addresses for the broker are advertized. You only need to set this if
	
	1. Your host has more than one active network interface, and
	
	2. You want to restrict client fail-over to a specific interface or interfaces.
	
	Each broker in the cluster is specified using the following form:
	
	`url = ["amqp:"][ user ["/" password] "@" ] protocol_addr`
	`("," protocol_addr)*`
	`protocol_addr = tcp_addr / rmda_addr / ssl_addr / ...`
	`tcp_addr = ["tcp:"] host [":" port]`
	`rdma_addr = "rdma:" host [":" port]`
	`ssl_addr = "ssl:" host [":" port]`
	
	In most cases, only one address is advertized, but more than one address can be specified in if the machine running the broker has more than one network interface card, and you want to allow clients to connect using multiple network interfaces. Use a comma delimiter (",") to separate brokers in the URL. Examples:
	
	• **amqp:tcp:192.168.1.103:5672** advertizes a single address to the broker for failover.
	
	• **amqp:tcp:192.168.1.103:5672,tcp:192.168.1.105:5672** advertizes two different addresses to the broker for failover, on two different network interfaces.

--cluster-cman	CMAN protects against the "split-brain" condition, in which a network failure splits the cluster into two sub-clusters that cannot communicate with each other. When "split-brain" occurs, each of the sub-clusters can access shared resources without knowledge of the other sub-cluster, resulting in corrupted cluster integrity. To avoid "split-brain", CMAN uses the notion of a "quorum". If more than half the cluster nodes are active, the cluster has quorum and can act. If half (or fewer) nodes are active, the cluster does not have quorum, and all cluster activity is stopped. There are other ways to define the quorum for particular use cases (e.g. a cluster of only 2 members), see the CMAN Wiki [http://sources.redhat.com/cluster/wiki] for more detail.

Options for High Availability Messaging Cluster	
	When enabled, the broker will wait until it belongs to a quorate cluster before accepting client connections. It continually monitors the quorum status and shuts down immediately if the node it runs on loses touch with the quorum.
--cluster-username	SASL username for connections between brokers.
--cluster-password	SASL password for connections between brokers.
--cluster-mechanism	SASL authentication mechanism for connections between brokers

If a broker is unable to establish a connection to another broker in the cluster, the log will contain SASL errors, e.g:

```
2009-aug-04 10:17:37 info SASL: Authentication failed: SASL(-13): user not found:
```

You can set the SASL user name and password used to connect to other brokers using the **cluster-username** and **cluster-password** properties when you start the broker. In most environment, it is easiest to create an account with the same user name and password on each broker in the cluster, and use these as the **cluster-username** and **cluster-password**. You can also set the SASL mode using **cluster-mechanism**. Remember that any mechanism you enable for broker-to-broker communication can also be used by a client, so do not enable **cluster-mechanism=ANONYMOUS** in a secure environment.

Once the cluster is running, run **qpid-cluster** to make sure that the brokers are running as one cluster. See the following section for details.

If the cluster is correctly configured, queues and messages are replicated to all brokers in the cluster, so an easy way to test the cluster is to run a program that routes messages to a queue on one broker, then to a different broker in the same cluster and read the messages to make sure they have been replicated. The **drain** and **spout** programs can be used for this test.

1.8.2. qpid-cluster

qpid-cluster is a command-line utility that allows you to view information on a cluster and its brokers, disconnect a client connection, shut down a broker in a cluster, or shut down the entire cluster. You can see the options using the **--help** option:

```
$ ./qpid-cluster --help


Usage:  qpid-cluster [OPTIONS] [broker-addr]

    broker-addr is in the form:   [username/password@] hostname | ip-address [:<po
    ex:  localhost, 10.1.1.7:10000, broker-host:10000, guest/guest@localhost

    Options:
    -C [--all-connections]  View client connections to all cluster members
    -c [--connections] ID   View client connections to specified member
    -d [--del-connection] HOST:PORT
    Disconnect a client connection
    -s [--stop] ID          Stop one member of the cluster by its ID
    -k [--all-stop]         Shut down the whole cluster
    -f [--force]            Suppress the 'are-you-sure?' prompt
    -n [--numeric]          Don't resolve names
```

Let's connect to a cluster and display basic information about the cluser and its brokers. When you connect to the cluster using **qpid-tool**, you can use the host and port for any broker in the cluster. For instance, if a broker in the cluster is running on `localhost` on port 6664, you can start **qpid-tool** like this:

```
$ qpid-cluster localhost:6664
```

Here is the output:

```
Cluster Name: local_test_cluster
Cluster Status: ACTIVE
Cluster Size: 3
Members: ID=127.0.0.1:13143 URL=amqp:tcp:192.168.1.101:6664,tcp:192.168.122.
: ID=127.0.0.1:13167 URL=amqp:tcp:192.168.1.101:6665,tcp:192.168.122.1:6665,
: ID=127.0.0.1:13192 URL=amqp:tcp:192.168.1.101:6666,tcp:192.168.122.1:6666,
```

The ID for each broker in cluster is given on the left. For instance, the ID for the first broker in the cluster is **127.0.0.1:13143**. The URL in the output is the broker's advertized address. Let's use the ID to shut the broker down using the **--stop** command:

```
$ ./qpid-cluster localhost:6664 --stop 127.0.0.1:13143
```

1.8.3. Failover in Clients

If a client is connected to a broker, the connection fails if the broker crashes or is killed. If heartbeat is enabled for the connection, a connection also fails if the broker hangs, the machine the broker is running on fails, or the network connection to the broker is lost — the connection fails no later than twice the heartbeat interval.

When a client's connection to a broker fails, any sent messages that have been acknowledged to the sender will have been replicated to all brokers in the cluster, any received messages that have not yet been acknowledged by the receiving client requeued to all brokers, and the client API notifies the application of the failure by throwing an exception.

Clients can be configured to automatically reconnect to another broker when it receives such an exception. Any messages that have been sent by the client, but not yet acknowledged as delivered, are resent. Any messages that have been read by the client, but not acknowledged, are delivered to the client.

TCP is slow to detect connection failures. A client can configure a connection to use a heartbeat to detect connection failure, and can specify a time interval for the heartbeat. If heartbeats are in use, failures will be detected no later than twice the heartbeat interval. The Java JMS client enables hearbeat by default. See the sections on Failover in Java JMS Clients and Failover in C++ Clients for the code to enable heartbeat.

1.8.3.1. Failover in Java JMS Clients

In Java JMS clients, client failover is handled automatically if it is enabled in the connection. Any messages that have been sent by the client, but not yet acknowledged as delivered, are resent. Any messages that have been read by the client, but not acknowledged, are sent to the client.

You can configure a connection to use failover using the **failover** property:

```
connectionfactory.qpidConnectionfactory = amqp://guest:guest@clientid/test?broker
```

This property can take three values:

Failover Modes

failover_exchange	If the connection fails, fail over to any other broker in the cluster.
roundrobin	If the connection fails, fail over to one of the brokers specified in the **brokerlist**.
singlebroker	Failover is not supported; the connection is to a single broker only.

In a Connection URL, heartbeat is set using the **idle_timeout** property, which is an integer corresponding to the heartbeat period in seconds. For instance, the following line from a JNDI properties file sets the heartbeat time out to 3 seconds:

```
connectionfactory.qpidConnectionfactory = amqp://guest:guest@clientid/test?broker
```

1.8.3.2. Failover and the Qpid Messaging API

The Qpid Messaging API also supports automatic reconnection in the event a connection fails. . Senders can also be configured to replay any in-doubt messages (i.e. messages whice were sent but not acknowleged by the broker. See "Connection Options" and "Sender Capacity and Replay" in *Programming in Apache Qpid* for details.

In C++ and python clients, heartbeats are disabled by default. You can enable them by specifying a heartbeat interval (in seconds) for the connection via the 'heartbeat' option.

See "Cluster Failover" in *Programming in Apache Qpid* for details on how to keep the client aware of cluster membership.

1.8.4. Error handling in Clusters

If a broker crashes or is killed, or a broker machine failure, broker connection failure, or a broker hang is detected, the other brokers in the cluster are notified that it is no longer a member of the cluster. If a new broker is joined to the cluster, it synchronizes with an active broker to obtain the current cluster state; if this synchronization fails, the new broker exit the cluster and aborts.

If a broker becomes extremely busy and stops responding, it stops accepting incoming work. All other brokers continue processing, and the non-responsive node caches all AIS traffic. When it resumes, the broker completes processes all cached AIS events, then accepts further incoming work.

Broker hangs are only detected if the watchdog plugin is loaded and the **--watchdog-interval** option is set. The watchdog plug-in kills the qpidd broker process if it becomes stuck for longer than the watchdog interval. In some cases, e.g. certain phases of error resolution, it is possible for a stuck process to hang other cluster members that are waiting for it to send a message. Using the watchdog, the stuck process

is terminated and removed from the cluster, allowing other members to continue and clients of the stuck process to fail over to other members.

Redundancy can also be achieved directly in the AIS network by specifying more than one network interface in the AIS configuration file. This causes Totem to use a redundant ring protocol, which makes failure of a single network transparent.

Redundancy can be achieved at the operating system level by using NIC bonding, which combines multiple network ports into a single group, effectively aggregating the bandwidth of multiple interfaces into a single connection. This provides both network load balancing and fault tolerance.

If any broker encounters an error, the brokers compare notes to see if they all received the same error. If not, the broker removes itself from the cluster and shuts itself down to ensure that all brokers in the cluster have consistent state. For instance, a broker may run out of disk space; if this happens, the broker shuts itself down. Examining the broker's log can help determine the error and suggest ways to prevent it from occuring in the future.

1.8.5. Persistence in High Availability Message Clusters

Persistence and clustering are two different ways to provide reliability. Most systems that use a cluster do not enable persistence, but you can do so if you want to ensure that messages are not lost even if the last broker in a cluster fails. A cluster must have all transient or all persistent members, mixed clusters are not allowed. Each broker in a persistent cluster has it's own independent replica of the cluster's state it its store.

1.8.5.1. Clean and Dirty Stores

When a broker is an active member of a cluster, its store is marked "dirty" because it may be out of date compared to other brokers in the cluster. If a broker leaves a running cluster because it is stopped, it crashes or the host crashes, its store continues to be marked "dirty".

If the cluster is reduced to a single broker, its store is marked "clean" since it is the only broker making updates. If the cluster is shut down with the command `qpid-cluster -k` then all the stores are marked clean.

When a cluster is initially formed, brokers with clean stores read from their stores. Brokers with dirty stores, or brokers that join after the cluster is running, discard their old stores and initialize a new store with an update from one of the running brokers. The **--truncate** option can be used to force a broker to discard all existing stores even if they are clean. (A dirty store is discarded regardless.)

Discarded stores are copied to a back up directory. The active store is in <data-dir>/rhm. Back-up stores are in <data-dir>/_cluster.bak.<nnnn>/rhm, where <nnnn> is a 4 digit number. A higher number means a more recent backup.

1.8.5.2. Starting a persistent cluster

When starting a persistent cluster broker, set the cluster-size option to the number of brokers in the cluster. This allows the brokers to wait until the entire cluster is running so that they can synchronize their stored state.

The cluster can start if:

- all members have empty stores, or
- at least one member has a clean store

All members of the new cluster will be initialized with the state from a clean store.

1.8.5.3. Stopping a persistent cluster

To cleanly shut down a persistent cluster use the command **qpid-cluster -k**. This causes all brokers to synchronize their state and mark their stores as "clean" so they can be used when the cluster restarts.

1.8.5.4. Starting a persistent cluster with no clean store

If the cluster has previously had a total failure and there are no clean stores then the brokers will fail to start with the log message `Cannot recover, no clean store.` If this happens you can start the cluster by marking one of the stores "clean" as follows:

1. Move the latest store backup into place in the brokers data-directory. The backups end in a 4 digit number, the latest backup is the highest number.

```
cd <data-dir>
mv rhm rhm.bak
cp -a _cluster.bak.<nnnn>/rhm .
```

2. Mark the store as clean:

```
qpid-cluster-store -c <data-dir>
```

Now you can start the cluster, all members will be initialized from the store you marked as clean.

1.8.5.5. Isolated failures in a persistent cluster

A broker in a persistent cluster may encounter errors that other brokers in the cluster do not; if this happens, the broker shuts itself down to avoid making the cluster state inconsistent. For example a disk failure on one node will result in that node shutting down. Running out of storage capacity can also cause a node to shut down because because the brokers may not run out of storage at exactly the same point, even if they have similar storage configuration. To avoid unnecessary broker shutdowns, make sure the queue policy size of each durable queue is less than the capacity of the journal for the queue.

1.9. Producer Flow Control

1.9.1. Overview

As of release 0.10, the C++ broker supports the use of flow control to throttle back message producers that are at risk of overflowing a destination queue.

Each queue in the C++ broker has two threshold values associated with it:

Flow Stop Threshold: this is the level of queue resource utilization above which flow control will be enabled. Once this threshold is crossed, the queue is considered in danger of overflow.

Flow Resume Threshold - this is the level of queue resource utilization below which flow control will be disabled. Once this threshold is crossed, the queue is no longer considered in danger of overflow.

In the above description, queue resource utilization may be defined as the total count of messages currently enqueued, or the total sum of all message content in bytes.

The value for a queue's Flow Stop Threshold must be greater than or equal to the value of the queue's Flow Resume Threshold.

1.9.1.1. Example

Let's consider a queue with a maximum limit set on the total number of messages that may be enqueued to that queue. Assume this maximum message limit is 1000 messages. Assume also that the user configures a Flow Stop Threshold of 900 messages, and a Flow Resume Threshold of 500 messages. Then the following holds:

The queue's initial flow control state is "OFF".

While the total number of enqueued messages is less than or equal to 900, the queue's flow control state remains "OFF".

When the total number of enqueued messages is greater than 900, the queue's flow control state transitions to "ON".

When the queue's flow control state is "ON", it remains "ON" until the total number of enqueued messages is less than 500. At that point, the queue's flow control state transitions to "OFF".

A similar example using total enqueued content bytes as the threshold units are permitted.

Thresholds may be set using both total message counts and total byte counts. In this case, the following rules apply:

1) Flow control is "ON" when either stop threshold value is crossed.

2) Flow control remains "ON" until both resume thresholds are satisfied.

1.9.1.2. Example

Let's consider a queue with a maximum size limit of 10K bytes, and 5000 messages. A user may assign a Flow Stop Threshold based on a total message count of 4000 messages. They may also assigne a Flow Stop Threshold of 8K bytes. The queue's flow control state transitions to "ON" if either threshold is crossed: (total-msgs greater-than 4000 OR total-bytes greater-than 8K).

Assume the user has assigned Flow Resume threshold's of 3000 messages and 6K bytes. Then the queue's flow control will remain active until both thresholds are satified: (total-msg less-than 3000 AND total-bytes less-than 6K).

The Broker enforces flow control by delaying the completion of the Message.Transfer command that causes a message to be delivered to a queue with active flow control. The completion of the Message.Transfer command is held off until flow control state transitions to "OFF" for all queues that are a destination for that command.

A message producing client is permitted to have a finite number of commands pending completion. When the total number of these outstanding commands reaches the limit, the client must not issue further commands until one or more of the outstanding commands have completed. This window of outstanding commands is considered the sender's "capacity". This allows any given producer to have a "capacity's" worth of messages blocked due to flow control before the sender must stop sending further messages.

This capacity window must be considered when determining a suitable flow stop threshold for a given queue, as a producer may send its capacity worth of messages _after_ a queue has reached the flow stop threshold. Therefore, a flow stop threshould should be set such that the queue can accomodate more messages without overflowing.

For example, assume two clients, C1 and C2, are producing messages to one particular destination queue. Assume client C1 has a configured capacity of 50 messages, and client C2's capacity is 15 messages. In this example, assume C1 and C2 are the only clients queuing messages to a given queue. If this queue has a Flow Stop Threshold of 100 messages, then, worst-case, the queue may receive up to 165 messages before clients C1 and C2 are blocked from sending further messages. This is due to the fact that the queue will enable flow control on receipt of its 101'st message - preventing the completion of the Message.Transfer command that carried the 101'st message. However, C1 and C2 are allowed to have a total of 65 (50 for C1 and 15 for C2) messages pending completion of Message.Transfer before they will stop producing messages. Thus, up to 65 messages may be enqueued beyond the flow stop threshold before the producers will be blocked.

1.9.2. User Interface

By default, the C++ broker assigns a queue's flow stop and flow resume thresholds when the queue is created. The C++ broker also allows the user to manually specify the flow control thresholds on a per queue basis.

However, queues that have been configured with a Limit Policy of type RING or RING-STRICT do NOT have queue flow thresholds enabled by default. The nature of a RING queue defines its behavior when its capacity is reach: replace the oldest message.

The flow control state of a queue can be determined by the "flowState" boolean in the queue's QMF management object. The queue's management object also contains a counter that increments each time flow control becomes active for the queue.

The broker applies a threshold ratio to compute a queue's default flow control configuration. These thresholds are expressed as a percentage of a queue's maximum capacity. There is one value for determining the stop threshold, and another for determining the resume threshold. The user may configure these percentages using the following broker configuration options:

```
--default-flow-stop-threshold ("Queue capacity level at which flow control
--default-flow-resume-threshold ("Queue capacity level at which flow contr
```

For example:

```
qpidd --default-flow-stop-threshold=90 --default-flow-resume-threshold=75
```

Sets the default flow stop threshold to 90% of a queue's maximum capacity and the flow resume threshold to 75% of the maximum capacity. If a queue is created with a default-queue-limit of 10000 bytes, then the default flow stop threshold would be 90% of 10000 = 9000 bytes and the flow resume threshold would be 75% of 10000 = 7500. The same computation is performed should a queue be created with a maximum size expressed as a message count instead of a byte count.

If not overridden by the user, the value of the default-flow-stop-threshold is 80% and the value of the default-flow-resume-threshold is 70%.

The user may disable default queue flow control broker-wide by specifying the value 0 for both of these configuration options. Note that flow control may still be applied manually on a per-queue basis in this case.

The user may manually set the flow thresholds when creating a queue. The following options may be provided when adding a queue using the **qpid-config** command line tool:

```
--flow-stop-size=N  Sets the queue's flow stop threshold to N total bytes.
--flow-resume-size=N  Sets the queue's flow resume threshold to N total by
--flow-stop-count=N Sets the queue's flow stop threshold to N total messag
--flow-resume-count=N Sets the queue's flow resume threshold to N total me
```

Flow thresholds may also be specified in the **queue.declare** method, via the **arguments** parameter map. The following keys can be provided in the arguments map for setting flow thresholds:

Table 1.10. Queue Declare Method Flow Control Arguments

Key	Value
qpid.flow_stop_size	integer - queue's flow stop threshold value in bytes
qpid.flow_resume_size	integer - queue's flow resume threshold value in bytes
qpid.flow_stop_count	integer - queue's flow stop threshold value as a message count
qpid.flow_resume_count	integer - queue's flow resume threshold value as a message count

The user may disable flow control on a per queue basis by setting the flow-stop-size and flow-stop-count to zero for the queue.

The current state of flow control for a given queue can be determined by the "flowStopped" statistic. This statistic is available in the queue's QMF management object. The value of flowStopped is True when the queue's capacity has exceeded the flow stop threshold. The value of flowStopped is False when the queue is no longer blocking due to flow control.

A queue will also track the number of times flow control has been activated. The "flowStoppedCount" statistic is incremented each time the queue's capacity exceeds a flow stop threshold. This statistic can be used to monitor the activity of flow control for any given queue over time.

Table 1.11. Flow Control Statistics available in Queue's QMF Class

Statistic Name	Type	Description
flowStopped	Boolean	If true, producers are blocked by flow control.
flowStoppedCount	count32	Number of times flow control was activated for this queue

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 1.12. 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 1.13. 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

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

1.11.1. SASL

1.11.1.1. 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 1.14. SASL Mechanism Support

Component	ANONYMOUS	CRAM-MD5	DIGEST-MD5	EXTERNAL	GSSAPI/ Kerberos	PLAIN
C++ Broker	M3[Section 1.1.1, “ Standard Mechanisms ” [47]] [47]	M3[Section 1.1.1, “ Standard Mechanisms ” [47]] [47]			M3[Section 1.1.1, “ Standard Mechanisms ” [47]] [47]	M1

		Mechanisms ” [47]]			Mechanisms ” [47]]	
C++ Client	M3[See 1, “ Standard Mechanisms ” [47]]					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

1.11.1.2. Custom Mechanisms

There have been some custom mechanisms added to our implementations.

Table 1.15. 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	

1.11.1.2.1. AMQPLAIN

1.11.1.2.2. 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).

1.12. Using Message Groups

1.12.1. Overview

The broker allows messaging applications to classify a set of related messages as belonging to a group. This allows a message producer to indicate to the consumer that a group of messages should be considered a single logical operation with respect to the application.

The broker can use this group identification to enforce policies controlling how messages from a given group can be distributed to consumers. For instance, the broker can be configured to guarantee all the messages from a particular group are processed in order across multiple consumers.

For example, assume we have a shopping application that manages items in a virtual shopping cart. A user may add an item to their shopping cart, then change their mind and remove it. If the application sends an *add* message to the broker, immediately followed by a *remove* message, they will be queued in the proper order - *add*, followed by *remove*.

However, if there are multiple consumers, it is possible that once a consumer acquires the *add* message, a different consumer may acquire the *remove* message. This allows both messages to be processed in parallel, which could result in a "race" where the *remove* operation is incorrectly performed before the *add* operation.

1.12.2. Grouping Messages

In order to group messages, the application would designate a particular message header as containing a message's *group identifier*. The group identifier stored in that header field would be a string value set by the message producer. Messages from the same group would have the same group identifier value. The key that identifies the header must also be known to the message consumers. This allows the consumers to determine a message's assigned group.

The header that is used to hold the group identifier, as well as the values used as group identifiers, are totally under control of the application.

1.12.3. The Role of the Broker

The broker will apply the following processing on each grouped message:

- Enqueue a received message on the destination queue.
- Determine the message's group by examining the message's group identifier header.
- Enforce *consumption ordering* among messages belonging to the same group.

Consumption ordering means that the broker will not allow outstanding unacknowledged messages to *more than one consumer for a given group*.

This means that only one consumer can be processing messages from a particular group at a given time. When the consumer acknowledges all of its acquired messages, then the broker *may* pass the next pending message from that group to a different consumer.

Specifically, for any given group the broker allows only the first N messages in the group to be delivered to a consumer. The value of N would be determined by the selected consumer's configured prefetch capacity. The broker blocks access by any other consumer to any remaining undelivered messages in that group. Once the receiving consumer has:

- acknowledged,

- released, or
- rejected

all the delivered messages, the broker allows the next messages in the group to be delivered. The next messages *may* be delivered to a different consumer.

Note well that distinct message groups would not block each other from delivery. For example, assume a queue contains messages from two different message groups - say group "A" and group "B" - and they are enqueued such that "A"'s messages are in front of "B". If the first message of group "A" is in the process of being consumed by a client, then the remaining "A" messages are blocked, but the messages of the "B" group are available for consumption by other consumers - even though it is "behind" group "A" in the queue.

1.12.4. Well Behaved Consumers

The broker can only enforce policy when delivering messages. To guarantee that strict message ordering is preserved, the consuming application must adhere to the following rules:

- completely process the data in a received message before accepting that message
- acknowledge (or reject) messages in the same order as they are received
- avoid releasing messages (see below)

The term *processed* means that the consumer has finished updating all application state affected by the message that has been received. See section 2.6.2. Transfer of Responsibility, of the AMQP-0.10 specification for more detail.

Be Advised

If a consumer does not adhere to the above rules, it may affect the ordering of grouped messages even when the broker is enforcing consumption order. This can be done by selectively acknowledging and releasing messages from the same group.

Assume a consumer has received two messages from group "A", "A-1" and "A-2", in that order. If the consumer releases "A-1" then acknowledges "A-2", "A-1" will be put back onto the queue and "A-2" will be removed from the queue. This allows another consumer to acquire and process "A-1" *after* "A-2" has been processed.

Under some application-defined circumstances, this may be acceptable behavior. However, if order must be preserved, the client should either release *all* currently held messages, or discard the target message using reject.

1.12.5. Broker Configuration

In order for the broker to determine a message's group, the key for the header that contains the group identifier must be provided to the broker via configuration. This is done on a per-queue basis, when the queue is first configured.

This means that message group classification is determined by the message's destination queue.

Specifically, the queue "holds" the header key that is used to find the message's group identifier. All messages arriving at the queue are expected to use the same header key for holding the identifer. Once the message is enqueued, the broker looks up the group identifier in the message's header, and classifies the message by its group.

Message group support can be enabled on a queue using the **qpid-config** command line tool. The following options should be provided when adding a new queue:

Table 1.16. qpid-config options for creating message group queues

Option	Description
--group-header=*header-name*	Enable message group support for this queue. Specify name of application header that holds the group identifier.
--shared-groups	Enforce ordered message group consumption across multiple consumers.

Message group support may also be specified in the **queue.declare** method via the **arguments** parameter map, or using the messaging address syntax. The following keys must be provided in the arguments map to enable message group support on a queue:

Table 1.17. Queue Declare/Address Syntax Message Group Configuration Arguments

Key	Value
qpid.group_header_key	string - key for message header that holds the group identifier value
qpid.shared_msg_group	1 - enforce ordering across multiple consumers

It is important to note that there is no need to provide the actual group identifer values that will be used. The broker learns this values as messages are recieved. Also, there is no practical limit - aside from resource limitations - to the number of different groups that the broker can track at run time.

Restrictions

Message grouping is not supported on LVQ or Priority queues.

Example 1.2. Creating a message group queue via qpid-config

This example uses the qpid-config tool to create a message group queue called "MyMsgQueue". The message header that contains the group identifier will use the key "GROUP_KEY".

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

Example 1.3. Creating a message group queue using address syntax (C++)

This example uses the messaging address syntax to create a message group queue with the same configuration as the previous example.

```
sender = session.createSender("MyMsgQueue;"
                              " {create:always, delete:receiver,"
                              " node: {x-declare: {arguments:"
                              " {'qpid.group_header_key':'GROUP_KEY',"
                              " 'qpid.shared_msg_group':1}}}}")
```

1.12.5.1. Default Group

Should a message without a group identifier arrive at a queue configured for message grouping, the broker assigns the message to the default group. Therefore, all such "unidentified" messages are considered by the broker as part of the same group. The name of the default group is **"qpid.no-group"**. This default can be overridden by suppling a different value to the broker configuration item **"default-message-group"**:

Example 1.4. Overriding the default message group identifier for the broker

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

1.13. Active-passive Messaging Clusters (Preview)

1.13.1. Overview

This release provides a preview of a new module for High Availability (HA). The new module is not yet complete or ready for production use, it being made available so that users can experiment with the new approach and provide feedback early in the development process. Feedback should go to user@qpid.apache.org [mailto:user@qpid.apache.org].

The old cluster module takes an *active-active* approach, i.e. all the brokers in a cluster are able to handle client requests simultaneously. The new HA module takes an *active-passive, hot-standby* approach.

In an active-passive cluster, only one broker, known as the *primary*, is active and serving clients at a time. The other brokers are standing by as *backups*. Changes on the primary are immediately replicated to all the backups so they are always up-to-date or "hot". If the primary fails, one of the backups is promoted to be the new primary. Clients fail-over to the new primary automatically. If there are multiple backups, the backups also fail-over to become backups of the new primary.

The new approach depends on an external *cluster resource manager* to detect failure of the primary and choose the new primary. The first supported resource manager will be rgmanager [https://fedorahosted.org/cluster/wiki/RGManager], but it will be possible to add integration with other resource managers in the future. The preview version is not integrated with any resource manager, you can use the **qpid-ha** tool to simulate the actions of a resource manager or do your own integration.

1.13.1.1. Why the new approach?

The new active-passive approach has several advantages compared to the existing active-active cluster module.

- It does not depend directly on openais or corosync. It does not use multicast which simplifies deployment.
- It is more portable: in environments that don't support corosync, it can be integrated with a resource manager available in that environment.
- Replication to a *disaster recovery* site can be handled as simply another node in the cluster, it does not require a separate replication mechanism.
- It can take advantage of features provided by the resource manager, for example virtual IP addresses.

- Improved performance and scalability due to better use of multiple CPU s

1.13.1.2. Limitations

There are a number of known limitations in the current preview implementation. These will be fixed in the production versions.

- Transactional changes to queue state are not replicated atomically. If the primary crashes during a transaction, it is possible that the backup could contain only part of the changes introduced by a transaction.
- During a fail-over one backup is promoted to primary and any other backups switch to the new primary. Messages sent to the new primary before all the backups have switched could be lost if the new primary itself fails before all the backups have switched.
- When used with a persistent store: if the entire cluster fails, there are no tools to help identify the most recent store.
- Acknowledgments are confirmed to clients before the message has been dequeued from replicas or indeed from the local store if that is asynchronous.
- A persistent broker must have its store erased before joining an existing cluster. In the production version a persistent broker will be able to load its store and avoid downloading messages that are in the store from the primary.
- Configuration changes (creating or deleting queues, exchanges and bindings) are replicated asynchronously. Management tools used to make changes will consider the change complete when it is complete on the primary, it may not yet be replicated to all the backups.
- Deletions made immediately after a failure (before all the backups are ready) may be lost on a backup. Queues, exchange or bindings that were deleted on the primary could re-appear if that backup is promoted to primary on a subsequent failure.
- Better control is needed over which queues/exchanges are replicated and which are not.
- There are some known issues affecting performance, both the throughput of replication and the time taken for backups to fail-over. Performance will improve in the production version.
- Federated links from the primary will be lost in fail over, they will not be re-connected on the new primary. Federation links to the primary can fail over.
- Only plain FIFO queues can be replicated. LVQ and ring queues are not yet supported.

1.13.2. Configuring the Brokers

The broker must load the `ha` module, it is loaded by default when you start a broker. The following broker options are available for the HA module.

Table 1.18. Options for High Availability Messaging Cluster

Options for High Availability Messaging Cluster	
--ha-cluster *yes|no*	Set to "yes" to have the broker join a cluster.
--ha-brokers *URL*	URL use by brokers to connect to each other. The URL lists the addresses of all the brokers in the cluster [a] in the following form:

Options for High Availability Messaging Cluster	

	`url = ["amqp:"][ user ["/" password] "@" ] addr ("," addr)*`
	`addr = tcp_addr / rmda_addr / ssl_addr / ...`
	`tcp_addr = ["tcp:"] host [":" port]`
	`rdma_addr = "rdma:" host [":" port]`
	`ssl_addr = "ssl:" host [":" port]'`

--ha-public-brokers *URL*	URL used by clients to connect to the brokers in the same format as **--ha-brokers** above. Use this option if you want client traffic on a different network from broker replication traffic. If this option is not set, clients will use the same URL as brokers.
--ha-username *USER* **--ha-password *PASS*** **--ha-mechanism *MECH***	Brokers use *USER*, *PASS*, *MECH* to authenticate when connecting to each other.

[a] If the resource manager supports virtual IP addresses then the URL can contain just the single virtual IP.

To configure a cluster you must set at least **ha-cluster** and **ha-brokers**

1.13.3. Creating replicated queues and exchanges

To create a replicated queue or exchange, pass the argument **qpid.replicate** when creating the queue or exchange. It should have one of the following three values:

- *all*: Replicate the queue or exchange, messages and bindings.
- *configuration*: Replicate the existence of the queue or exchange and bindings but don't replicate messages.
- *none*: Don't replicate, this is the default.

Bindings are automatically replicated if the queue and exchange being bound both have replication argument of **all** or **confguration**, they are not replicated otherwise. You can create replicated queues and exchanges with the **qpid-config** management tool like this:

```
qpid-config add queue myqueue --replicate all
```

To create replicated queues and exchangs via the client API, add a **node** entry to the address like this:

```
"myqueue;{create:always,node:{x-declare:{arguments:{'qpid.replicate':all}}}}
```

1.13.4. Client Fail-over

Clients can only connect to the single primary broker. All other brokers in the cluster are backups, and they automatically reject any attempt by a client to connect.

Clients are configured with the addreses of all of the brokers in the cluster. [1] When the client tries to connect initially, it will try all of its addresses until it successfully connects to the primary. If the primary fails, clients will try to try to re-connect to all the known brokers until they find the new primary.

Suppose your cluster has 3 nodes: **node1**, **node2** and **node3** all using the default AMQP port.

With the C++ client, you specify all the cluster addresses in a single URL, for example:

```
qpid::messaging::Connection c("node1:node2:node3");
```

With the python client, you specify **reconnect=True** and a list of *host:port* addresses as **reconnect_urls** when calling **establish** or **open**

```
connection = qpid.messaging.Connection.establish("node1", reconnect=True, "reconn
```

1.13.5. Broker fail-over

Broker fail-over is managed by a *cluster resource manager*. The initial preview version of HA is not integrated with a resource manager, the production version will be integrated with rgmanager [https://fedorahosted.org/cluster/wiki/RGManager] and it may be integrated with other resource managers in the future.

The resource manager is responsible for ensuring that there is exactly one broker is acting as primary at all times. It selects the initial primary broker when the cluster is started, detects failure of the primary, and chooses the backup to promote as the new primary.

You can simulate the actions of a resource manager, or indeed do your own integration with a resource manager using the **qpid-ha** tool. The command

```
qpid-ha promote -b host:port
```

will promote the broker listening on *host*:*port* to be the primary. You should only promote a broker to primary when there is no other primary in the cluster. The brokers will not detect multiple primaries, they rely on the resource manager to do that.

A clustered broker always starts initially in *discovery* mode. It uses the addresses configured in the **ha-brokers** configuration option and tries to connect to each in turn until it finds to the primary. The resource manager is responsible for choosing on of the backups to promote as the initial primary.

If the primary fails, all the backups are disconnected and return to discovery mode. The resource manager chooses one to promote as the new primary. The other backups will eventually discover the new primary and reconnect.

1.13.6. Broker Administration

You can connect to a backup broker with the administrative tool **qpid-ha**. You can also connect with the tools **qpid-config**, **qpid-route** and **qpid-stat** if you pass the flag **--ha-admin** on the command line. If

[1] If the resource manager supports virtual IP addresses then the clients can be configured with a single virtual IP address.

you do connect to a backup you should not modify any of the replicated queues, as this will disrupt the replication and may result in message loss.

1.14. Queue Replication with the HA module

As well as support for an active-passive cluster, the `ha` module also allows you to replicate individual queues. The *original* queue is used as normal. The *replica* queue is updated automatically as messages are added to or removed from the original queue.

To create a replica you need the HA module to be loaded on both the orignal and replica brokers. Note that it is not safe to modify the replica queue other than via the automatic updates from the original. Adding or removing messages on the replica queue will make replication inconsistent and may cause message loss. The HA module does *not* enforce restricted access to the replica queue (as it does in the case of a cluster) so it is up to the application to ensure the replca is not used until it has been disconnected from the original.

Suppose that **myqueue** is a queue on **node1** and we want to create a replica of **myqueue** on **node2** (where both brokers are using the default AMQP port.) This is accomplished by the command:

```
qpid-config --broker=node2 add queue --start-replica node1 myqueue
```

If **myqueue** already exists on the replica broker you can start replication from the original queue like this:

```
qpid-ha replicate -b node2 node1 myqueue
```

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
- qpid-ha - used to interact with the High Availability module

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!

2.1.5. Using qpid-ha

This utility lets you monitor and control the activity of the clustering behavior provided by the HA module.

```
qpid-ha --help
usage: qpid-ha <command> [<arguments>]

Commands are:

  ready        Test if a backup broker is ready.
  query        Print HA configuration settings.
  set          Set HA configuration settings.
  promote      Promote broker from backup to primary.
  replicate    Set up replication from <queue> on <remote-broker> to <queue> on th

For help with a command type: qpid-ha <command> --help
```

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

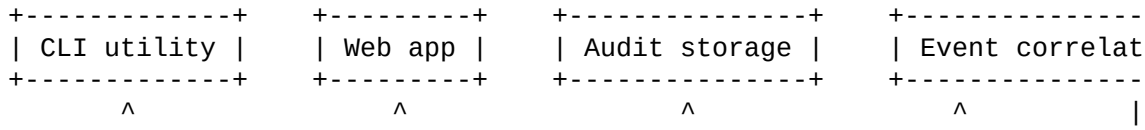
This section introduces important concepts underlying QMF.

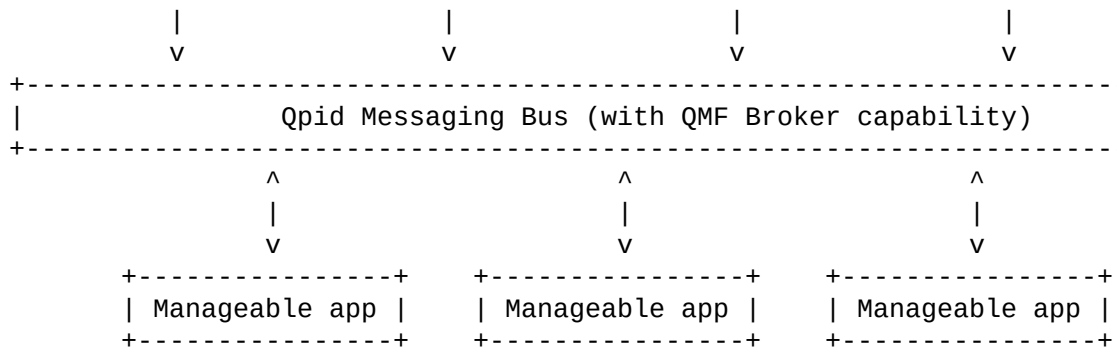
2.2.3.1. Console, Agent, and Broker

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.





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

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