AMQP Messaging Broker (Implemented in C++)

AMQP Messaging Broker (Implemented in C++)

Table of Contents

Introduction vii
1. Running the AMQP Messaging Broker 1
1.1. Running a Qpid C++ Broker 1
1.1.1. Building the C++ Broker and Client Libraries 1
1.1.2. Running the C++ Broker 1
1.1.3. Most common questions getting qpidd running 1
1.1.4. Authentication 2
1.1.5. Slightly more complex configuration 3
1.1.6. Loading extra modules 5
1.1.7. Timestamping Received Messages 6
1.1.8. Logging Options 7
1.2. Cheat Sheet for configuring Queue Options 11
1.2.1. Configuring Queue Options 11
1.3. Cheat Sheet for configuring Exchange Options 13
1.3.1. Configuring Exchange Options 13
1.4. Broker Federation 14
1.4.1. Message Routes 15
1.4.2. Federation Topologies 16
1.4.3. Federation among High Availability Message Clusters 16
1.4.4. The qpid-route Utility 17
1.4.5. Broker options affecting federation 23
1.5. Security 23
1.5.1. User Authentication 24
1.5.2. Authorization 27
1.5.3. User Connection and Queue Quotas 47
1.5.4. Encryption using SSL 51
1.6. LVQ - Last Value Queue 53
1.6.1. Understanding LVQ 53
1.6.2. Creating a Last Value Queue 54
1.6.3. LVQ Example 55
1.6.4. Deprecated LVQ Modes 55
1.7. Queue State Replication 56
1.7.1. Asynchronous Replication of Queue State 56
1.8. Producer Flow Control 59
1.8.1. Overview 59
1.8.2. User Interface 60
1.9. AMQP compatibility 62
1.9.1. AMQP Compatibility of Qpid releases: 63
1.9.2. Interop table by AMQP specification version 63
1.10. Qpid Interoperability Documentation 64
1.10.1. SASL 64
1.11. Using Message Groups 65
1.11.1. Overview 65
1.11.2. Grouping Messages 66
1.11.3. The Role of the Broker 66
1.11.4. Well Behaved Consumers 67
1.11.5. Broker Configuration 67
1.12. Active-Passive Messaging Clusters 69
1.12.1. Overview 69
1.12.2. Virtual IP Addresses 70
1.12.3. Configuring the Brokers 70

1.12.4. The Cluster Resource Manager 72
1.12.5. Configuring with **rgmanager** as resource manager 72
1.12.6. Broker Administration Tools 75
1.12.7. Controlling replication of queues and exchanges 75
1.12.8. Client Connection and Fail-over 76
1.12.9. Security and Access Control. 78
1.12.10. Integrating with other Cluster Resource Managers 79
1.12.11. Using a message store in a cluster 79
1.12.12. Troubleshooting a cluster 79
1.13. Replicating Queues with the HA module 82
1.13.1. Replicating queues 82
1.13.2. Replicating queues between clusters 83
2. Managing the AMQP Messaging Broker 84
2.1. Managing the C++ Broker 84
2.1.1. Using qpid-config 84
2.1.2. Using qpid-route 86
2.1.3. Using qpid-tool 87
2.1.4. Using qpid-printevents 91
2.1.5. Using qpid-ha 91
2.2. Qpid Management Framework 91
2.2.1. What Is QMF 92
2.2.2. Getting Started with QMF 92
2.2.3. QMF Concepts 92
2.2.4. The QMF Protocol 96
2.2.5. How to Write a QMF Console 97
2.2.6. How to Write a QMF Agent 97
2.3. QMF Python Console Tutorial 97
2.3.1. Prerequisite - Install Qpid Messaging 97
2.3.2. Synchronous Console Operations 98
2.3.3. Asynchronous Console Operations 102
2.3.4. Discovering what Kinds of Objects are Available 106

List of Tables

1.1. QMF Management - Broker Methods for Managing the Timestamp Configuration 6
1.2. C++ Broker Log Severity Levels 7
1.3. C++ Broker Log Categories 7
1.4. C++ Broker Log Statement Attributes 8
1.5. C++ Broker Log Enable/Disable RULE Format 8
1.6. C++ Broker Log Enable/Disable Settings Tables 8
1.7. C++ Broker Log Statement Visibility Determination 9
1.8. QMF Management - Broker Methods for Managing the Log Enable/Disable Settings 9
1.9. **qpid-route** options 17
1.10. State values in **$ qpid-route list connections** 23
1.11. Broker Options for Federation 23
1.12. ACL Rules: permission 29
1.13. ACL Rules: action 29
1.14. ACL Rules:object 29
1.15. ACL Rules: property 30
1.16. Broker Lookup Events With Allowed Action, Object, and Properties 32
1.17. ACL User Name and Domain Name Substitution Keywords 36
1.18. Topic Exchange Wildcard Match Examples 38
1.19. SSL Client Environment Variables for C++ clients 52
1.20. Queue Declare Method Flow Control Arguments 61
1.21. Flow Control Statistics available in Queue's QMF Class 62
1.22. AMQP Version Support by Qpid Release 63
1.23. AMQP Version Support - alternate format 63
1.24. SASL Mechanism Support 64
1.25. SASL Custom Mechanisms 65
1.26. qpid-config options for creating message group queues 67
1.27. Queue Declare/Address Syntax Message Group Configuration Arguments 68
1.28. Broker Options for High Availability Messaging Cluster 71
1.29. HA Security Options 78
2.1. XML Attributes for QMF Properties and Statistics 94
2.2. QMF Datatypes 95
2.3. XML Schema Mapping for QMF Types 96
2.4. QMF Python Console Class Methods 103

List of Examples

1.1. Enabling Message Timestamping via QMF - Python 6
1.2. Querying Log Settings via qpid-ctrl utility 10
1.3. Setting Log Settings via qpid-ctrl utility 10
1.4. Creating a message group queue via qpid-config 68
1.5. Creating a message group queue using address syntax (C++) 68
1.6. Overriding the default message group identifier for the broker 69

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 C++ 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:
  -t [ --trace ]              Enables all logging
  --log-enable RULE (notice+) Enables logging for selected levels and components.
                              RULE is in the form 'LEVEL[+-][:PATTERN]'
                              LEVEL is one of:
                                 trace debug info notice warning error critical
                              PATTERN is a logging category name, or a namespace-q
                              function name or name fragment.
                                 Logging category names are:
                                 Security Broker Management Protocol System HA Mes
                                 Network Test Client Model Unspecified

                              For example:
                                  '--log-enable warning+'
                                  logs all warning, error and critical messages.

                                  '--log-enable trace+:Broker'
                                  logs all category 'Broker' messages.

                                  '--log-enable debug:framing'
                                  logs debug messages from all functions with 'fra
                                  the namespace or function name.

                              This option can be used multiple times

  --log-disable RULE          Disables logging for selected levels and components.
                              RULE is in the form 'LEVEL[+-][:PATTERN]'
                              LEVEL is one of:
                                 trace debug info notice warning error critical
                              PATTERN is a logging category name, or a namespace-q
                              function name or name fragment.
                                 Logging category names are:
                                 Security Broker Management Protocol System HA Mes
                                 Network Test Client Model Unspecified

                              For example:
                                  '--log-disable warning-'
                                  disables logging all warning, notice, info, debu
                                  trace messages.

                                  '--log-disable trace:Broker'
                                  disables all category 'Broker' trace messages.

                                  '--log-disable debug-:qmf::'
```

```
                                    disables logging debug and trace messages from a
                                    with 'qmf::' in the 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
  --log-hires-timestamp yes|no (0)      Use hi-resolution timestamps in log
                                        messages
  --log-category yes|no (1)             Include category in log messages
  --log-prefix STRING                   Prefix to prepend to all log messages

Logging sink options:
  --log-to-stderr yes|no (1)            Send logging output to stderr
  --log-to-stdout yes|no (0)            Send logging output to stdout
  --log-to-file FILE                    Send log output to FILE.
  --log-to-syslog yes|no (0)            Send logging output to syslog;
                                        customize using --syslog-name and
                                        --syslog-facility
  --syslog-name NAME (qpidd)            Name to use in syslog messages
  --syslog-facility LOG_XXX (LOG_DAEMON)
                                        Facility to use in syslog 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")
```

1.1.8. Logging Options

The C++ Broker provides a rich set of logging options. To use logging effectively a user must select a useful set of options to expose the log messages of interest. This section introduces the logging options and how they are used in practice.

1.1.8.1. Logging Concepts

1.1.8.1.1. Log Level

The C++ Broker has a traditional set of log severity levels. The log levels range from low frequency and high importance critical level to high frequency and low importance trace level.

Table 1.2. C++ Broker Log Severity Levels

Name	Level
critical	high
error	
warning	
notice	
info	
debug	
trace	low

1.1.8.1.2. Log Category

The C++ Broker groups log messages into categories. The log category name may then be used to enable and disable groups of related messages at varying log levels.

Table 1.3. C++ Broker Log Categories

Name
Security
Broker
Management
Protocol
System
HA
Messaging
Store
Network
Test
Client
Model
Unspecified

Generally speaking the log categories are groupings of messages from files related by thier placement in the source code directory structure. The *Model* category is an exception. Debug log entries identified by the Model category expose the creation, deletion, and usage statistics for managed objects in the broker. Log messages in the Model category are emitted by source files scattered throughout the source tree.

1.1.8.1.3. Log Statement Attributes

Every log statement in the C++ Broker has fixed attributes that may be used in enabling or disabling log messages.

Table 1.4. C++ Broker Log Statement Attributes

Name	Description
Level	Severity level
Category	Category
Function	Namespace-qualified source function name

1.1.8.2. Enabling and Disabling Log Messages

The Qpid C++ Broker has hundreds of log message statements in the source code. Under typical conditions most of the messages are deselected and never emitted as actual logs. However, under some circumstances debug and trace messages must be enabled to analyze broker behavior. This section discusses how the broker enables and disables log messages.

At startup the broker processes command line and option file '--log-enable RULE' and '--log-disable RULE' options using the following rule format:

```
LEVEL[+-][:PATTERN}
```

Table 1.5. C++ Broker Log Enable/Disable RULE Format

Name	Description
LEVEL	Severity level
[+-]	Option level modifiers. *'+'* indicates *this level and above*. *'-'* indicates *this level and below*.
[:PATTERN]	If PATTERN matches a Category name then the log option applies only to log messages with the named category. Otherwise, the pattern is stored as a function name match string.

As the options are procesed the results are aggregated into two pairs of tables.

Table 1.6. C++ Broker Log Enable/Disable Settings Tables

Name	Description
Function Table	A set of vectors of accumulated function name patterns. There is a separate vector of name patterns for each log level.
Category Table	A simple two dimensional array of boolean values indexed by [Level][Category] indicating if all log

Name	Description
	statements are enabled for the Level and Category pair.

--log-enable statements and --log-disable statements are aggregated into dedicated Function and Category tables. With this scheme multiple conflicting log enable and disable commands may be processed in any order yet produce consistent patterns of enabled broker log statements.

1.1.8.3. Determining if a Log Statement is Enabled

Function Table Lookups are simple string pattern matches where the searchable text is the domain-name qualified function name from the log statement and the search pattern is the set of Function Table entries for a given log level.

Category Table Lookups are boolean array queries where the Level and Category indexes are from the log statement.

Each log statment sends its Level, Category, and FunctionName to the Logger for evaluation. As a result the log statement is either visible or hidden.

Table 1.7. C++ Broker Log Statement Visibility Determination

Test	Description
Disabled Function	If the statement matches a Disabled Function pattern then the statement is hidden.
Disabled Category	If the Disabled Category table for this [Level] [Category] is true then the statement is hidden.
Enabled Function	If the statement matches a Enabled Function pattern then the statement is visible.
Enabled Category	If the Enabled Category table for this [Level] [Category] is true then the statement is visible.
Unreferenced	Log statements that are unreferenced by specific enable rules are by default hidden.

1.1.8.4. Changing Log Enable/Disable Settings at Run Time

The C++ Broker provides QMF management methods that allow users to query and to set the log enable and disable settings while the broker is running.

Table 1.8. QMF Management - Broker Methods for Managing the Log Enable/ Disable Settings

Method	Description
getLogLevel	Get the log enable/disable settings.
setLogLevel	Set the log enable/disable settings.

The management methods use a RULE format similar to the option RULE format:

```
[!]LEVEL[+-][:PATTERN]
```

The difference is the leading exclamation point that identifies disable rules.

Example 1.2. Querying Log Settings via qpid-ctrl utility

At start up a C++ Broker may have the following options:

```
--log-enable debug+
--log-enable trace+:Protocol
--log-disable info-:Management
```

The following command:

```
qpid-ctrl getLogLevel
```

will return the following result:

```
level=debug+,trace+:Protocol,!info-:Management
```

Example 1.3. Setting Log Settings via qpid-ctrl utility

New broker log options may be set at any time using qpid-ctrl

```
qpid-ctrl setLogLevel level='debug+:Broker !debug-:broker::Broker::ManagementMet
```

1.1.8.5. Discovering Log Sources

A common condition for a user is being swamped by log messages that are not interesting for some debug situation. Conversely, a particular log entry may be of interest all the time but enabling all log levels just to see a single log entry is too much. How can a user find and specify a pattern to single out logs of interest?

The easiest way to hide messages it to disable logs at log level and category combinations. This may not always work since using only these coarse controls the log messages of interest may also be hidden. To discover a more precise filter to specify the messages you want to show or to hide you may temporarily enable the *"--log-function=yes"* option. The following log entries show a typical log message without and with the log function names enabled:

```
2013-05-01 11:16:01 [Broker] notice Broker running
2013-05-01 11:16:54 [Broker] notice qpid::broker::Broker::run: Broker running
```

This log entry is emitted by function *qpid::broker::Broker::run* and this is the function name pattern to be used in specific log enable and disable rules. For example, this log entry could be disabled with any of the following:

```
--log-disable notice                            [1]
--log-disable notice:qpid::                     [2]
```

```
--log-disable notice:Broker                     [3]
--log-disable notice-:Broker::run               [4]
--log-disable notice:qpid::broker::Broker::run  [5]
```

- [1] Disables all messages at notice level.
- [2] Disables all messages at notice level in qpid:: name space. This is very broad and disables many log messages.
- [3] Disables the category *[Broker]* and is not specific to the function. Category names supercede function name fragments in log option processing
- [4] Disables the function.
- [5] Disables the function.

Remember that the log filter matching PATTERN strings are matched against the domain-name qualified function names associated with the log statement and not against the log message text itself. That is, in the previous example log filters cannot be set on the log text *Broker running*

1.2. Cheat Sheet for configuring Queue Options

1.2.1. Configuring Queue Options

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

- Section 1.2.1, “ Configuring Queue Options ”
- • Section 1.2.1.1, “ Applying Queue Sizing Constraints ”
  - Section 1.2.1.2, “ Changing the Queue ordering Behaviors (FIFO/LVQ) ”
  - Section 1.2.1.3, “ Setting additional behaviors ”
  - • ???
  - Section 1.2.1.4, “ Other Clients ”

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

- Section 1.8, “ Producer Flow Control ”

1.2.1.1. Applying Queue Sizing Constraints

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

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

- REJECT - Reject the published message
- FLOW_TO_DISK - Flow the messages to disk, to preserve memory
- RING - start overwriting messages in a ring based on sizing. If head meets tail, advance head

- RING_STRICT - start overwriting messages in a ring based on sizing. If head meets tail, AND the consumer has the tail message acquired it will reject

Examples:

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

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

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

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

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

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

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

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

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

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

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

Example:

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

    QueueOptions qo;
    qo.setOrdering(LVQ);

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

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

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

Notes:

- Messages that are dequeued and the re-queued will have the following exceptions. a.) if a new message has been queued with the same key, the re-queue from the consumer, will combine these two messages.

b.) If an update happens for a message of the same key, after the re-queue, it will not update the re-queued message. This is done to protect a client from being able to adversely manipulate the queue.

- Acquire: When a message is acquired from the queue, no matter it's position, it will behave the same as a dequeue
- LVQ does not support durable messages. If the queue or messages are declared durable on an LVQ, the durability will be ignored.

A fully worked Section 1.6.3, “LVQ Example” can be found here

1.2.1.3. Setting additional behaviors

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> <routin
qpid-route [OPTIONS] route del <dest-broker> <src-broker> <exchange> <routin

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

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

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 1.10. 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.

1.4.5. Broker options affecting federation

The following broker options affect federation:

Table 1.11. Broker Options for Federation

Options for Federation	
`federation-tag` *`NAME`*	A unique name to identify this broker in federation network. If not specified, the broker will generate a unique identifier.
`link-maintenance-interval` *`SECONDS`* [b]	Interval to check if links need to be re-connected. Default 2 seconds. Can be a sub-second interval for faster failover, e.g. 0.1 seconds.
`link-heartbeat-interval` *`SECONDS`* [b]	Heart-beat interval for federation links. If no heart-beat is received for twice the interval the link is considered dead. Default 120 seconds.

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 1.5.4, “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
```

Alternatively the ACL file may use *allow mode* by placing:

```
acl allow all all
```

as the final line in the ACL file. In *allow mode* all actions by all users are allowed unless otherwise denied by specific ACL rules. The ACL rule which selects *deny mode* or *allow mode* must be the last line in the ACL rule file.

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
```

An ACL file can define per user connection and queue quotas:

```
group admin ted@QPID martin@QPID
group blacklist usera@qpid userb@qpid
quota connections 10 admin
quota connections  5 all
quota connections  0 blacklist
quota queues      50 admin
quota queues       5 all
quota queues       1 test@qpid
```

Performance Note: Most ACL queries are performed infrequently. The overhead associated with ACL passing an allow or deny decision on the creation of a queue is negligible compared to actually creating and using the queue. One notable exception is the **publish exchange** query. ACL files with no *publish exchange* rules are noted and the broker short circuits the logic associated with the per-messsage *publish exchange* ACL query. However, if an ACL file has any *publish exchange* rules then the broker is required to perform a *publish exchange* query for each message published. Users with performance critical applications are encouraged to structure exchanges, queues, and bindings so that the *publish exchange* ACL rules are unnecessary.

1.5.2.1. ACL Syntax

ACL rules follow this syntax:

```
aclline = ( comment | aclspec | groupspec | quotaspec )

comment = "#" [ STRING ]

aclspec = "acl" permission ( groupname | name | "all" )
          ( action | "all" ) [ ( object | "all" ) [ ( property "=" STRING )* ] ]

groupspec = "group" groupname ( name )* [ "\" ]

groupcontinuation = ( name )* [ "\" ]

quotaspec = "quota" ( "connections" | "queues" ) NUMBER ( groupname | name | "all"

name = ( ALPHANUMERIC | "-" | "_" | "." | "@" | "/" ) [ ( ALPHANUMERIC | "-" | "_"

groupname = ( ALPHANUMERIC | "-" | "_" ) [ ( ALPHANUMERIC | "-" | "_" )* ]

permission = "allow" | "allow-log" | "deny" | "deny-log"

action = "consume" | "publish" | "create" | "access" |
         "bind"    | "unbind"  | "delete" | "purge"  |
         "update"

object = "queue"  | "exchange" | "broker"     | "link" |
         "method" | "query"    | "connection"
```

```
property =  "name" | "durable" | "routingkey" | "autodelete" |
            "exclusive" | "type" | "alternate" | "queuename"  |
            "exchangename" | "schemapackage" | "schemaclass" |
            "policytype" | "paging" |
            "queuemaxsizelowerlimit"  | "queuemaxsizeupperlimit" |
            "queuemaxcountlowerlimit" | "queuemaxcountupperlimit" |
            "filemaxsizelowerlimit"   | "filemaxsizeupperlimit" |
            "filemaxcountlowerlimit"  | "filemaxcountupperlimit" |
            "pageslowerlimit"         | "pagesupperlimit" |
            "pagefactorlowerlimit"    | "pagefactorupperlimit"
```

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 1.12. 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 1.13. ACL Rules: action

access	Accessing or reading an object
bind	Associating a queue to an exchange with a routing key.
consume	Using an object
create	Creating an object.
delete	Deleting an object.
move	Moving messages between queues.
publish	Authenticating an incoming message.
purge	Purging a queue.
redirect	Redirecting messages between queues
reroute	Rerouting messages from a queue to an exchange
unbind	Disassociating a queue from an exchange with a routing key.
update	Changing a broker configuration setting.

Table 1.14. ACL Rules:object

broker	
connection	Incoming TCP/IP connection
exchange	
link	A federation or inter-broker link

method	Management method
query	Management query of an object or class
queue	

Table 1.15. ACL Rules: property

Property	Type	Description	Usage
name	String	Rule refers to objects with this name. When 'name' is blank or absent then the rule applies to all objects of the given type.	
alternate	String	Name of an alternate exchange	CREATE QUEUE, CREATE EXCHANGE, ACCESS QUEUE, ACCESS EXCHANGE, DELETE QUEUE, DELETE EXCHANGE
autodelete	Boolean	Indicates whether or not the object gets deleted when the connection that created it is closed	CREATE QUEUE, CREATE EXCHANGE, ACCESS QUEUE, ACCESS EXCHANGE, DELETE QUEUE
durable	Boolean	Rule applies to durable objects	CREATE QUEUE, CREATE EXCHANGE, ACCESS QUEUE, ACCESS EXCHANGE, DELETE QUEUE, DELETE EXCHANGE
exchangename	String	Name of the exchange to which queue's entries are routed	REROUTE QUEUE
filemaxcountlowerlimit	Integer	Minimum value for file.max_count (files)	CREATE QUEUE
filemaxcountupperlimit	Integer	Maximum value for file.max_count (files)	CREATE QUEUE
filemaxsizelowerlimit	Integer	Minimum value for file.max_size (64kb pages)	CREATE QUEUE
filemaxsizeupperlimit	Integer	Maximum value for file.max_size (64kb pages)	CREATE QUEUE
host	String	Target TCP/IP host or host range for create connection rules	CREATE CONNECTION
exclusive	Boolean	Indicates the presence of an `exclusive` flag	CREATE QUEUE, ACCESS QUEUE, DELETE QUEUE

Property	Type	Description	Usage
pagefactorlowerlimit	Integer	Minimum value for size of a page in paged queue	CREATE QUEUE
pagefactorupperlimit	Integer	Maximum value for size of a page in paged queue	CREATE QUEUE
pageslowerlimit	Integer	Minimum value for number of paged queue pages in memory	CREATE QUEUE
pagesupperlimit	Integer	Maximum value for number of paged queue pages in memory	CREATE QUEUE
paging	Boolean	Indicates if the queue is a paging queue	CREATE QUEUE
policytype	String	"ring", "self-destruct", "reject"	CREATE QUEUE, ACCESS QUEUE, DELETE QUEUE
queuename	String	Name of the target queue	ACCESS EXCHANGE, BIND EXCHANGE, MOVE QUEUE, UNBIND EXCHANGE
queuemaxsizelowerlimit	Integer	Minimum value for queue.max_size (memory bytes)	CREATE QUEUE, ACCESS QUEUE
queuemaxsizeupperlimit	Integer	Maximum value for queue.max_size (memory bytes)	CREATE QUEUE, ACCESS QUEUE
queuemaxcountlowerlimit	Integer	Minimum value for queue.max_count (messages)	CREATE QUEUE, ACCESS QUEUE
queuemaxcountupperlimit	Integer	Maximum value for queue.max_count (messages)	CREATE QUEUE, ACCESS QUEUE
routingkey	String	Specifies routing key	BIND EXCHANGE, UNBIND EXCHANGE, ACCESS EXCHANGE, PUBLISH EXCHANGE
schemaclass	String	QMF schema class name	ACCESS METHOD, ACCESS QUERY
schemapackage	String	QMF schema package name	ACCESS METHOD
type	String	Type of exchange, such as topic, fanout, or xml	CREATE EXCHANGE, ACCESS EXCHANGE, DELETE EXCHANGE

1.5.2.1.1. ACL Action-Object-Property Combinations

Not every ACL action is applicable to every ACL object. Furthermore, not every property may be specified for every action-object pair. The following table lists the broker events that trigger ACL lookups. Then for each event it lists the action, object, and properties allowed in the lookup.

User-specified ACL rules constrain property sets to those that match one or more of the action and object pairs. For example these rules are allowed:

```
acl allow all access exchange
acl allow all access exchange name=abc
acl allow all access exchange name=abc durable=true
```

These rules could possibly match one or more of the broker lookups. However, this rule is not allowed:

```
acl allow all access exchange queuename=queue1 durable=true
```

Properties *queuename* and *durable* are not in the list of allowed properties for any 'access exchange' lookup. This rule would never match a broker lookup query and would never contribute to an allow or deny decision.

For more information about matching ACL rules please refer to ACL Rule Matching

Table 1.16. Broker Lookup Events With Allowed Action, Object, and Properties

Lookup Event	Action	Object	Properties
User querying message timestamp setting	access	broker	
AMQP 0-10 protocol received 'query'	access	exchange	name
AMQP 0-10 query binding	access	exchange	name queuename routingkey
AMQP 0-10 exchange declare	access	exchange	name type alternate durable autodelete
AMQP 1.0 exchange access	access	exchange	name type durable
AMQP 1.0 node resolution	access	exchange	name
Management method request	access	method	name schemapackage schemaclass
Management agent method request	access	method	name schemapackage schemaclass
Management agent query	access	query	name schemaclass
QMF 'query queue' method	access	queue	name
AMQP 0-10 query	access	queue	name
AMQP 0-10 queue declare	access	queue	name alternate durable exclusive autodelete policytype queuemaxcountlowerlimit

Lookup Event	Action	Object	Properties
			queuemaxcountupperlimit queuemaxsizelowerlimit queuemaxsizeupperlimit
AMQP 1.0 queue access	access	queue	name alternate durable exclusive autodelete policytype queuemaxcountlowerlimit queuemaxcountupperlimit queuemaxsizelowerlimit queuemaxsizeupperlimit
AMQP 1.0 node resolution	access	queue	name
AMQP 0-10 or QMF bind request	bind	exchange	name queuename routingkey
AMQP 1.0 new outgoing link from exchange	bind	exchange	name queuename routingkey
AMQP 0-10 subscribe request	consume	queue	name
AMQP 1.0 new outgoing link from queue	consume	queue	name
TCP/IP connection creation	create	connection	host
Create exchange	create	exchange	name type alternate durable autodelete
Interbroker link creation	create	link	
Interbroker link creation	create	link	
Create queue	create	queue	name alternate durable exclusive autodelete policytype paging pageslowerlimit pagesupperlimit pagefactorlowerlimit pagefactorupperlimit queuemaxcountlowerlimit queuemaxcountupperlimit queuemaxsizelowerlimit queuemaxsizeupperlimit filemaxcountlowerlimit filemaxcountupperlimit filemaxsizelowerlimit filemaxsizeupperlimit
Delete exchange	delete	exchange	name type alternate durable
Delete queue	delete	queue	name alternate durable exclusive autodelete policytype

Lookup Event	Action	Object	Properties
Management 'move queue' request	move	queue	name queuename
AMQP 0-10 received message processing	publish	exchange	name routingkey
AMQP 1.0 establish sender link to queue	publish	exchange	routingkey
AMQP 1.0 received message processing	publish	exchange	name routingkey
Management 'purge queue' request	purge	queue	name
Management 'purge queue' request	purge	queue	name
Management 'redirect queue' request	redirect	queue	name queuename
Management 'reroute queue' request	reroute	queue	name exchangename
Management 'unbind exchange' request	unbind	exchange	name queuename routingkey
User modifying message timestamp setting	update	broker	

1.5.2.2. ACL Syntactic Conventions

1.5.2.2.1. Comments

- A line starting with the **#** character is considered a comment and is ignored.
- Embedded comments and trailing comments are not allowed. The **#** is commonly found in routing keys and other AMQP literals which occur naturally in ACL rule specifications.

1.5.2.2.2. White Space

- Empty lines and lines that contain only whitespace (' ', '\f', '\n', '\r', '\t', '\v') are ignored.
- Additional whitespace between and after tokens is allowed.
- Group and Acl definitions must start with **group** and **acl** respectively and with no preceding whitespace.

1.5.2.2.3. Character Set

- ACL files use 7-bit ASCII characters only
- Group names may contain only
  - **[a-z]**
  - **[A-Z]**
  - **[0-9]**

  - '-' hyphen
  - '_' underscore
- Individual user names may contain only
  - **[a-z]**
  - **[A-Z]**
  - **[0-9]**
  - '-' hyphen
  - '_' underscore
  - '.' period
  - '@' ampersand
  - '/' slash

1.5.2.2.4. Case Sensitivity

- All tokens are case sensitive. *`name1`* is not the same as *`Name1`* and *`create`* is not the same as *`CREATE`*.

1.5.2.2.5. Line Continuation

- Group lists can be extended to the following line by terminating the line with the '\' character. No other ACL file lines may be continued.
- Group specification lines may be continued only after the group name or any of the user names included in the group. See example below.
- Lines consisting solely of a '\' character are not permitted.
- The '\' continuation character is recognized only if it is the last character in the line. Any characters after the '\' are not permitted.

```
#
# Examples of extending group lists using a trailing '\' character
#
group group1 name1 name2 \
name3 name4 \
name5

group group2 \
            group1 \
            name6
#
# The following are illegal:
#
# '\' must be after group name
#
```

```
group \
      group3 name7 name8
#
# No empty extension line
#
group group4 name9 \
                   \
             name10
```

1.5.2.2.6. Line Length

- ACL file lines are limited to 1024 characters.

1.5.2.2.7. ACL File Keywords

ACL reserves several words for convenience and for context sensitive substitution.

1.5.2.2.7.1. The all Keyword

The keyword **all** is reserved. It may be used in ACL rules to match all individuals and groups, all actions, or all objects.

- acl allow all create queue
- acl allow bob@QPID all queue
- acl allow bob@QPID create all

1.5.2.2.7.2. User Name and Domain Name Keywords

In the C++ Broker 0.20 a simple set of user name and domain name substitution variable keyword tokens is defined. This provides administrators with an easy way to describe private or shared resources.

Symbol substitution is allowed in the ACL file anywhere that text is supplied for a property value.

In the following table an authenticated user named bob.user@QPID.COM has his substitution keywords expanded.

Table 1.17. ACL User Name and Domain Name Substitution Keywords

Keyword	Expansion
${userdomain}	bob_user_QPID_COM
${user}	bob_user
${domain}	QPID_COM

- The original user name has the period "." and ampersand "@" characters translated into underscore "_". This allows substitution to work when the substitution keyword is used in a routingkey in the Acl file.
- The Acl processing matches ${userdomain} before matching either ${user} or ${domain}. Rules that specify the combination ${user}_${domain} will never match.

```
# Example:
#
# Administrators can set up Acl rule files that allow every user to create a
# private exchange, a private queue, and a private binding between them.
```

```
# In this example the users are also allowed to create private backup exchanges,
# queues and bindings. This effectively provides limits to user's exchange,
# queue, and binding creation and guarantees that each user gets exclusive
# access to these resources.
#
#
# Create primary queue and exchange:
#
acl allow all create  queue    name=$\{user}-work alternate=$\{user}-work2
acl deny  all create  queue    name=$\{user}-work alternate=*
acl allow all create  queue    name=$\{user}-work
acl allow all create  exchange name=$\{user}-work alternate=$\{user}-work2
acl deny  all create  exchange name=$\{user}-work alternate=*
acl allow all create  exchange name=$\{user}-work
#
# Create backup queue and exchange
#
acl deny  all create  queue    name=$\{user}-work2 alternate=*
acl allow all create  queue    name=$\{user}-work2
acl deny  all create  exchange name=$\{user}-work2 alternate=*
acl allow all create  exchange name=$\{user}-work2
#
# Bind/unbind primary exchange
#
acl allow all bind   exchange name=$\{user}-work routingkey=$\{user} queuename=$
acl allow all unbind exchange name=$\{user}-work routingkey=$\{user} queuename=$
#
# Bind/unbind backup exchange
#
acl allow all bind   exchange name=$\{user}-work2 routingkey=$\{user} queuename=
acl allow all unbind exchange name=$\{user}-work2 routingkey=$\{user} queuename=
#
# Access primary exchange
#
acl allow all access exchange name=$\{user}-work routingkey=$\{user} queuename=$
#
# Access backup exchange
#
acl allow all access exchange name=$\{user}-work2 routingkey=$\{user} queuename=
#
# Publish primary exchange
#
acl allow all publish exchange name=$\{user}-work routingkey=$\{user}
#
# Publish backup exchange
#
acl allow all publish exchange name=$\{user}-work2 routingkey=$\{user}
#
# deny mode
#
acl deny all all
```

1.5.2.2.8. Wildcards

ACL prividesv two types of wildcard matching to provide flexibility in writing rules.

1.5.2.2.8.1. Property Value Wildcard

Text specifying a property value may end with a single trailing ***** character. This is a simple wildcard match indicating that strings which match up to that point are matches for the ACL property rule. An ACL rule such as

```
acl allow bob@QPID create queue name=bob*
```

allow user bob@QPID to create queues named bob1, bob2, bobQueue3, and so on.

1.5.2.2.8.2. Topic Routing Key Wildcard

In the C++ Broker 0.20 the logic governing the ACL Match has changed for each ACL rule that contains a routingkey property. The routingkey property is matched according to Topic Exchange match logic the broker uses when it distributes messages published to a topic exchange.

Routing keys are hierarchical where each level is separated by a period:

- weather.usa
- weather.europe.germany
- weather.europe.germany.berlin
- company.engineering.repository

Within the routing key hierarchy two wildcard characters are defined.

- ***** matches one field
- # matches zero or more fields

Suppose an ACL rule file is:

```
acl allow-log uHash1@COMPANY publish exchange name=X routingkey=a.#.b
acl deny all all
```

When user uHash1@COMPANY attempts to publish to exchange X the ACL will return these results:

Table 1.18. Topic Exchange Wildcard Match Examples

routingkey in publish to exchange X	result
a.b	allow-log
a.x.b	allow-log
a.x.y.zz.b	allow-log
a.b.	deny
q.x.b	deny

1.5.2.3. ACL Rule Matching

The minimum matching criteria for ACL rules are:

- An actor (individually named or group member)
- An action
- An object

If a rule does not match the minimum criteria then that rule does not control the ACL allow or deny decision.

ACL rules optionally specify object names and property name=value pairs. If an ACL rule specifies an object name or property values than all of them must match to cause the rule to match.

The following illustration shows how ACL rules are processed to find matching rules.

```
# Example of rule matching
#
# Using this ACL file content:

(1)  acl deny bob create exchange name=test durable=true passive=true
(2)  acl deny bob create exchange name=myEx type=direct
(3)  acl allow all all

#
# Lookup 1. id:bob action:create objectType:exchange name=test
#           {durable=false passive=false type=direct alternate=}
#
# ACL Match Processing:
#  1. Rule 1 passes minimum criteria with user bob, action create,
#     and object exchange.
#  2. Rule 1 matches name=test.
#  3. Rule 1 does not match the rule's durable=true with the requested
#     lookup of durable=false.
#  4. Rule 1 does not control the decision and processing continues
#     to Rule 2.
#  5. Rule 2 passes minimum criteria with user bob, action create,
#     and object exchange.
#  6. Rule 2 does not match the rule's name=myEx with the requested
#     lookup of name=test.
#  7. Rule 2 does not control the decision and processing continues
#     to Rule 3.
#  8. Rule 3 matches everything and the decision is 'allow'.
#
# Lookup 2. id:bob action:create objectType:exchange name=myEx
#           {durable=true passive=true type=direct alternate=}
#
# ACL Match Processing:
#  1. Rule 1 passes minimum criteria with user bob, action create,
#     and object exchange.
#  2. Rule 1 does not match the rule's name=test with the requested
#     lookup of name=myEx.
#  3. Rule 1 does not control the decision and processing continues
#     to Rule 2.
#  4. Rule 2 passes minimum criteria with user bob, action create,
#     and object exchange.
```

```
#  5. Rule 2 matches name=myEx.
#  6. Rule 2 matches the rule's type=direct with the requested
#     lookup of type=direct.
#  7. Rule 2 is the matching rule and the decision is 'deny'.
#
```

Referring to ACL Properties Allowed for each Action and Object table observe that some Action/Object pairs have different sets of allowed properties. For example different broker ACL lookups for *access exchange* have different property subsets.

```
[1] access exchange name
[2] access exchange name type alternate durable autodelete
[3] access exchange name queuename routingkey
[4] access exchange name type durable
```

If an ACL rule specifies the *autodelete* property then it can possibly match only the second case above. It can never match cases 1, 3, and 4 because the broker calls to ACL will not present the autodelete property for matching. To get proper matching the ACL rule must have only the properties of the intended lookup case.

```
acl allow bob access exchange alternate=other    ! may match pattern 2 only
acl allow bob access exchange queuename=other    ! may match pattern 3 only
acl allow bob access exchange durable=true       ! may match patterns 2 and 4
acl deny  bob access exchange                    ! may match all patterns
```

1.5.2.4. 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.

ACL allows specification of conflicting rules. Be sure to specify the most specific rules first followed by more general rules. Here is an example:

```
group users alice@QPID bob@QPID charlie@QPID
acl deny  charlie@QPID create queue
acl allow users        create queue
acl deny all all
```

In this example users alice and bob would be able to create queues due to their membership in the users group. However, user charlie is denied from creating a queue despite his membership in the users group because a deny rule for him is stated before the allow rule for the users group.

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
```

1.5.2.5. Auditing ACL Settings

The 0.30 C++ Broker ACL module provides a comprehensive set of run-time and debug logging checks. The following example ACL file is used to illustrate working with the ACL module debugging features.

```
group x a@QPID b@QPID b2@QPID b3@QPID
acl allow all delete broker
acl allow all create queue name=abc
acl allow all create queue exchangename=xyz
acl allow all create connection host=1.1.1.1
acl allow all access exchange alternate=abc queuename=xyz
acl allow all access exchange queuename=xyz
acl allow all access exchange alternate=abc
acl allow a@qpid all all exchangename=123
acl allow b@qpid all all
acl allow all all
```

When this file is loaded it will show the following (truncated, formatted) Info-level log.

```
notice ACL: Read file "/home/chug/acl/svn-acl.acl"
warning ACL rule ignored: Broker never checks for rules with
                          action: 'delete' and object: 'broker'
```

```
warning ACL rule ignored: Broker checks for rules with
                          action: 'create' and object: 'queue'
            but will never match with property set: { exchangename=xyz }
warning ACL rule ignored: Broker checks for rules with
                          action: 'access' and object: 'exchange'
            but will never match with property set: { alternate=abc queuename=xy
info ACL Plugin loaded
```

Three of the rules are invalid. The first invalid rule is rejected because there are no rules that specify 'delete broker' regardless of the properties. The other two rules are rejected because the property sets in the ACL rule don't match any broker lookups.

The ACL module only issues a warning about these rules and continues to operate. Users upgrading from previous versions should be concerned that these rules never had any effect and should fix the rules to have the property sets needed to allow or deny the intended broker events.

The next illustration shows the Debug-level log. Debug log level includes information about constructing the rule tables, expanding groups and keywords, connection and queue quotas, and connection black and white lists.

```
notice ACL: Read file "/home/chug/acl/svn-acl.acl"
debug ACL: Group list: 1 groups found:
debug ACL:   "x": a@QPID b2@QPID b3@QPID b@QPID
debug ACL: name list: 7 names found:
debug ACL:  * a@QPID a@qpid b2@QPID b3@QPID b@QPID b@qpid
debug ACL: Rule list: 10 ACL rules found:
debug ACL:    1 allow [*] delete broker
warning ACL rule ignored: Broker never checks for rules with
                          action: 'delete' and object: 'broker'
debug ACL:    2 allow [*] create queue name=abc
debug ACL:    3 allow [*] create queue exchangename=xyz
warning ACL rule ignored: Broker checks for rules with
                          action: 'create' and object: 'queue'
                    but will never match with property set: { exchangename=xyz }
debug ACL:    4 allow [*] create connection host=1.1.1.1
debug ACL:    5 allow [*] access exchange alternate=abc queuename=xyz
warning ACL rule ignored: Broker checks for rules with
                          action: 'access' and object: 'exchange'
                    but will never match with property set: { alternate=abc queue
debug ACL:    6 allow [*] access exchange queuename=xyz
debug ACL:    7 allow [*] access exchange alternate=abc
debug ACL:    8 allow [a@qpid] * * exchangename=123
debug ACL:    9 allow [b@qpid] * *
debug ACL:   10 allow [*] *
debug ACL: connections quota: 0 rules found:
debug ACL: queues quota: 0 rules found:
debug ACL: Load Rules
debug ACL: Processing 10 allow [*] *
debug ACL: FoundMode allow
debug ACL: Processing  9 allow [b@qpid] * *
debug ACL: Adding actions {access,bind,consume,create,delete,move,publish,purge,
                           redirect,reroute,unbind,update}
               to objects {broker,connection,exchange,link,method,query,queue}
               with props { }
```

```
                for users {b@qpid}
debug ACL: Processing  8 allow [a@qpid] * * exchangename=123
debug ACL: Adding actions {access,bind,consume,create,delete,move,publish,purge,
                           redirect,reroute,unbind,update}
               to objects {broker,connection,exchange,link,method,query,queue}
               with props { exchangename=123 }
                for users {a@qpid}
debug ACL: Processing  7 allow [*] access exchange alternate=abc
debug ACL: Adding actions {access}
               to objects {exchange}
               with props { alternate=abc }
                for users {*,a@QPID,a@qpid,b2@QPID,b3@QPID,b@QPID,b@qpid}
debug ACL: Processing  6 allow [*] access exchange queuename=xyz
debug ACL: Adding actions {access}
               to objects {exchange}
               with props { queuename=xyz }
                for users {*,a@QPID,a@qpid,b2@QPID,b3@QPID,b@QPID,b@qpid}
debug ACL: Processing  5 allow [*] access exchange alternate=abc queuename=xyz
debug ACL: Processing  4 allow [*] create connection host=1.1.1.1
debug ACL: Processing  3 allow [*] create queue exchangename=xyz
debug ACL: Processing  2 allow [*] create queue name=abc
debug ACL: Adding actions {create}
               to objects {queue}
               with props { name=abc }
                for users {*,a@QPID,a@qpid,b2@QPID,b3@QPID,b@QPID,b@qpid}
debug ACL: Processing  1 allow [*] delete broker
debug ACL: global Connection Rule list : 1 rules found :
debug ACL:    1 [ruleMode = allow {(1.1.1.1,1.1.1.1)}
debug ACL: User Connection Rule lists : 0 user lists found :
debug ACL: Transfer ACL is Enabled!
info ACL Plugin loaded
```

The previous illustration is interesting because it shows the settings as the *all* keywords are being expanded. However, that does not show the information about what is actually going into the ACL lookup tables.

The next two illustrations show additional information provided by Trace-level logs for ACL startup. The first shows a dump of the broker's internal action/object/properties table. This table is authoratative.

```
trace ACL: Definitions of action, object, (allowed properties) lookups
trace ACL: Lookup  1: "User querying message timestamp setting  "
                        access   broker     ()
trace ACL: Lookup  2: "AMQP 0-10 protocol received 'query'      "
                        access   exchange   (name)
trace ACL: Lookup  3: "AMQP 0-10 query binding                  "
                        access   exchange   (name,routingkey,queuename)
trace ACL: Lookup  4: "AMQP 0-10 exchange declare               "
                        access   exchange   (name,durable,autodelete,type,altern
trace ACL: Lookup  5: "AMQP 1.0 exchange access                 "
                        access   exchange   (name,durable,type)
trace ACL: Lookup  6: "AMQP 1.0 node resolution                 "
                        access   exchange   (name)
trace ACL: Lookup  7: "Management method request                "
                        access   method     (name,schemapackage,schemaclass)
```

```
trace ACL: Lookup  8: "Management agent method request          "
                        access   method     (name,schemapackage,schemaclass)
trace ACL: Lookup  9: "Management agent query                   "
                        access   query      (name,schemaclass)
trace ACL: Lookup 10: "QMF 'query queue' method                 "
                        access   queue      (name)
trace ACL: Lookup 11: "AMQP 0-10 query                          "
                        access   queue      (name)
trace ACL: Lookup 12: "AMQP 0-10 queue declare                  "
                        access   queue      (name,durable,autodelete,exclusive,a
                          policytype,queuemaxsizelowerlimit,queuemaxsizeupperlim
                          queuemaxcountlowerlimit,queuemaxcountupperlimit)
trace ACL: Lookup 13: "AMQP 1.0 queue access                    "
                        access   queue      (name,durable,autodelete,exclusive,a
                          policytype,queuemaxsizelowerlimit,queuemaxsizeupperlim
                          queuemaxcountlowerlimit,queuemaxcountupperlimit)
trace ACL: Lookup 14: "AMQP 1.0 node resolution                 "
                        access   queue      (name)
trace ACL: Lookup 15: "AMQP 0-10 or QMF bind request            "
                        bind     exchange   (name,routingkey,queuename)
trace ACL: Lookup 16: "AMQP 1.0 new outgoing link from exchange "
                        bind     exchange   (name,routingkey,queuename)
trace ACL: Lookup 17: "AMQP 0-10 subscribe request              "
                        consume  queue      (name)
trace ACL: Lookup 18: "AMQP 1.0 new outgoing link from queue    "
                        consume  queue      (name)
trace ACL: Lookup 19: "TCP/IP connection creation               "
                        create   connection (host)
trace ACL: Lookup 20: "Create exchange                          "
                        create   exchange   (name,durable,autodelete,type,altern
trace ACL: Lookup 21: "Interbroker link creation                "
                        create   link       ()
trace ACL: Lookup 22: "Interbroker link creation                "
                        create   link       ()
trace ACL: Lookup 23: "Create queue                             "
                        create   queue      (name,durable,autodelete,exclusive,
                          alternate,policytype,paging,
                          queuemaxsizelowerlimit,queuemaxsizeupperlimit,
                          queuemaxcountlowerlimit,queuemaxcountupperlimit,
                          filemaxsizelowerlimit,filemaxsizeupperlimit,
                          filemaxcountlowerlimit,filemaxcountupperlimit,
                          pageslowerlimit,pagesupperlimit,
                          pagefactorlowerlimit,pagefactorupperlimit)
trace ACL: Lookup 24: "Delete exchange                          "
                        delete   exchange   (name,durable,type,alternate)
trace ACL: Lookup 25: "Delete queue                             "
                        delete   queue      (name,durable,autodelete,exclusive,
                          alternate,policytype)
trace ACL: Lookup 26: "Management 'move queue' request          "
                        move     queue      (name,queuename)
trace ACL: Lookup 27: "AMQP 0-10 received message processing    "
                        publish  exchange   (name,routingkey)
trace ACL: Lookup 28: "AMQP 1.0 establish sender link to queue  "
                        publish  exchange   (routingkey)
```

```
trace ACL: Lookup 29: "AMQP 1.0 received message processing     "
                        publish  exchange   (name,routingkey)
trace ACL: Lookup 30: "Management 'purge queue' request         "
                        purge    queue      (name)
trace ACL: Lookup 31: "Management 'purge queue' request         "
                        purge    queue      (name)
trace ACL: Lookup 32: "Management 'redirect queue' request      "
                        redirect queue      (name,queuename)
trace ACL: Lookup 33: "Management 'reroute queue' request       "
                        reroute  queue      (name,exchangename)
trace ACL: Lookup 34: "Management 'unbind exchange' request     "
                        unbind   exchange   (name,routingkey,queuename)
trace ACL: Lookup 35: "User modifying message timestamp setting "
                        update   broker     ()
```

The final illustration shows a dump of every rule for every user in the ACL database. It includes the user name, action, object, original ACL rule number, allow or deny status, and a cross reference indicating which Lookup Events the rule could possibly satisfy.

Note that rules identified by *User:* * are the rules in effect for users otherwise unnamed in the ACL file.

```
trace ACL: Decision rule cross reference
trace ACL: User: b@qpid   access   broker
                      Rule: [rule 9 ruleMode = allow props{ }]
                            may match Lookups : (1)
trace ACL: User: *        access   exchange
                      Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                            may match Lookups : (3)
trace ACL: User: *        access   exchange
                      Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                            may match Lookups : (4)
trace ACL: User: a@QPID   access   exchange
                      Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                            may match Lookups : (3)
trace ACL: User: a@QPID   access   exchange
                      Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                            may match Lookups : (4)
trace ACL: User: a@qpid   access   exchange
                      Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                            may match Lookups : (3)
trace ACL: User: a@qpid   access   exchange
                      Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                            may match Lookups : (4)
trace ACL: User: b2@QPID  access   exchange
                      Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                            may match Lookups : (3)
trace ACL: User: b2@QPID  access   exchange
                      Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                            may match Lookups : (4)
trace ACL: User: b3@QPID  access   exchange
                      Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                            may match Lookups : (3)
trace ACL: User: b3@QPID  access   exchange
```

```
                        Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                              may match Lookups : (4)
trace ACL: User: b@QPID   access   exchange
                        Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                              may match Lookups : (3)
trace ACL: User: b@QPID   access   exchange
                        Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                              may match Lookups : (4)
trace ACL: User: b@qpid   access   exchange
                        Rule: [rule 6 ruleMode = allow props{ queuename=xyz }]
                              may match Lookups : (3)
trace ACL: User: b@qpid   access   exchange
                        Rule: [rule 7 ruleMode = allow props{ alternate=abc }]
                              may match Lookups : (4)
trace ACL: User: b@qpid   access   exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (2,3,4,5,6)
trace ACL: User: b@qpid   access   method
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (7,8)
trace ACL: User: b@qpid   access   query
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (9)
trace ACL: User: b@qpid   access   queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (10,11,12,13,14)
trace ACL: User: b@qpid   bind     exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (15,16)
trace ACL: User: b@qpid   consume  queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (17,18)
trace ACL: User: b@qpid   create   connection
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (19)
trace ACL: User: b@qpid   create   exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (20)
trace ACL: User: b@qpid   create   link
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (21,22)
trace ACL: User: *        create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: a@QPID   create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: a@qpid   create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: b2@QPID  create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: b3@QPID  create   queue
```

```
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: b@QPID   create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: b@qpid   create   queue
                        Rule: [rule 2 ruleMode = allow props{ name=abc }]
                              may match Lookups : (23)
trace ACL: User: b@qpid   create   queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (23)
trace ACL: User: b@qpid   delete   exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (24)
trace ACL: User: b@qpid   delete   queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (25)
trace ACL: User: b@qpid   move     queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (26)
trace ACL: User: b@qpid   publish  exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (27,28,29)
trace ACL: User: b@qpid   purge    queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (30,31)
trace ACL: User: b@qpid   redirect queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (32)
trace ACL: User: a@qpid   reroute  queue
                        Rule: [rule 8 ruleMode = allow props{ exchangename=123 }]
                              may match Lookups : (33)
trace ACL: User: b@qpid   reroute  queue
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (33)
trace ACL: User: b@qpid   unbind   exchange
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (34)
trace ACL: User: b@qpid   update   broker
                        Rule: [rule 9 ruleMode = allow props{ }]
                              may match Lookups : (35)
```

1.5.3. User Connection and Queue Quotas

The ACL module enforces various quotas and thereby limits user activity.

1.5.3.1. Connection Count Limits

The ACL module creates broker command line switches that set limits on the number of concurrent connections allowed per user or per client host address. These settings are not specified in the ACL file.

```
--max-connections           N
--connection-limit-per-user N
```

```
--connection-limit-per-ip   N
```

--max-connections specifies an upper limit for all user connections.

--connection-limit-per-user specifies an upper limit for each user based on the authenticated user name. This limit is enforced regardless of the client IP address from which the connection originates.

--connection-limit-per-ip specifies an upper limit for connections for all users based on the originating client IP address. This limit is enforced regardless of the user credentials presented with the connection.

- Note that addresses using different transports are counted separately even though the originating host is actually the same physical machine. In the setting illustrated above a host would allow N_IP connections from [::1] IPv6 transport localhost and another N_IP connections from [127.0.0.1] IPv4 transport localhost.
- The connection-limit-per-ip and connection-limit-per-user counts are active simultaneously. From a given client system users may be denied access to the broker by either connection limit.

The 0.22 C++ Broker ACL module accepts fine grained per-user connection limits through quota rules in the ACL file.

```
quota connections 10 admins userX@QPID
```

- User `all` receives the value passed by the command line switch `--connection-limit-per-user`.
- Values specified in the ACL rule for user `all` overwrite the value specified on the command line if any.
- Connection quotas values are determined by first searching for the authenticated user name. If that user name is not specified then the value for user `all` is used. If user `all` is not specified then the connection is denied.
- The connection quota values range from 0..65530 inclusive. A value of zero disables connections from that user.
- A user's quota may be specified many times in the ACL rule file. Only the last value specified is retained and enforced.
- Per-user connection quotas are disabled when two conditions are true: 1) No --connection-limit-per-user command line switch and 2) No `quota connections` rules in the ACL file. Per-user connections are always counted even if connection quotas are not enforced. This supports ACL file reloading that may subsequently enable per-user connection quotas.
- An ACL file reload may lower a user's connection quota value to a number lower than the user's current connection count. In that case the active connections remain unaffected. New connections are denied until that user closes enough of his connections so that his count falls below the configured limit.

1.5.3.2. Connection Limits by Host Name

The 0.30 C++ Broker ACL module adds the ability to create allow and deny lists of the TCP/IP hosts from which users may connect. The rule accepts these forms:

```
acl allow user create connection host=host1
acl allow user create connection host=host1,host2
acl deny  user create connection host=all
```

Using the form **host=host1** specifies a single host. With a single host the name may resolve to multiple TCP/IP addresses. For example *localhost* resolves to both *127.0.0.1* and *::1* and possibly many other addresses. A connection from any of the addresses associated with this host matches the rule and the connection is allowed or denied accordingly.

Using the form **host=host1,host2** specifies a range of TCP/IP addresses. With a host range each host must resolve to a single TCP/IP address and the second address must be numerically larger than the first. A connection from any host where host >= host1 and host <= host2 match the rule and the connection is allowed or denied accordingly.

Using the form **host=all** specifies all TCP/IP addresses. A connection from any host matches the rule and the connection is allowed or denied accordingly.

Connection denial is only applied to incoming TCP/IP connections. Other socket types are not subjected to nor denied by range checks.

Connection creation rules are divided into three categories:

1. User = all, host != all

   These define global rules and are applied before any specific user rules. These rules may be used to reject connections before any AMPQ protocol is run and before any user names have been negotiated.

2. User != all, host = any legal host or 'all'

   These define user rules. These rules are applied after the global rules and after the AMQP protocol has negotiated user identities.

3. User = all, host = all

   This rule defines what to do if no other rule matches. The default value is "ALLOW". Only one rule of this type may be defined.

The following example illustrates how this feature can be used.

```
group admins alice bob chuck
group Company1 c1_usera c1_userb
group Company2 c2_userx c2_usery c2_userz
acl allow admins   create connection host=localhost
acl allow admins   create connection host=10.0.0.0,10.255.255.255
acl allow admins   create connection host=192.168.0.0,192.168.255.255
acl allow admins   create connection host=[fc00::],[fc00::ff]
acl allow Company1 create connection host=company1.com
acl deny  Company1 create connection host=all
acl allow Company2 create connection host=company2.com
acl deny  Company2 create connection host=all
```

In this example admins may connect from localhost or from any system on the 10.0.0.0/24, 192.168.0.0/16, and fc00::/7 subnets. Company1 users may connect only from company1.com and Company2 users may

connect only from company2.com. However, this example has a flaw. Although the admins group has specific hosts from which it is allowed to make connections it is not blocked from connecting from anywhere. The Company1 and Company2 groups are blocked appropriately. This ACL file may be rewritten as follows:

```
group admins alice bob chuck
group Company1 c1_usera c1_userb
group Company2 c2_userx c2_usery c2_userz
acl allow admins   create connection host=localhost
acl allow admins   create connection host=10.0.0.0,10.255.255.255
acl allow admins   create connection host=192.168.0.0,192.168.255.255
acl allow admins   create connection host=[fc00::],[fc00::ff]
acl allow Company1 create connection host=company1.com
acl allow Company2 create connection host=company2.com
acl deny  all      create connection host=all
```

Now admins are blocked from connecting from anywhere but their allowed hosts.

1.5.3.3. Queue Limits

The ACL module creates a broker command line switch that set limits on the number of queues each user is allowed to create. This settings is not specified in the ACL file.

```
--max-queues-per-user N
```

The queue limit is set for all users on the broker.

The 0.22 C++ Broker ACL module accepts fine grained per-user queue limits through quota rules in the ACL file.

```
quota queues 10 admins userX@QPID
```

- User `all` receives the value passed by the command line switch `--max-queues-per-user`.
- Values specified in the ACL rule for user `all` overwrite the value specified on the command line if any.
- Queue quotas values are determined by first searching for the authenticated user name. If that user name is not specified then the value for user `all` is used. If user `all` is not specified then the queue creation is denied.
- The queue quota values range from 0..65530 inclusive. A value of zero disables queue creation by that user.
- A user's quota may be specified many times in the ACL rule file. Only the last value specified is retained and enforced.
- Per-user queue quotas are disabled when two conditions are true: 1) No --queue-limit-per-user command line switch and 2) No `quota queues` rules in the ACL file. Per-user queue creations are always counted even if queue quotas are not enforced. This supports ACL file reloading that may subsequently enable per-user queue quotas.

- An ACL file reload may lower a user's queue quota value to a number lower than the user's current queue count. In that case the active queues remain unaffected. New queues are denied until that user closes enough of his queues so that his count falls below the configured limit.

1.5.4. 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 `CN`literal>, and appending any `DC`literal>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 1.19. 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. Producer Flow Control

1.8.1. 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.

1.8.1.1. 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.

1.8.1.2. 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.

1.8.2. 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 1.20. Queue Declare Method Flow Control Arguments

Key	Value
qpid.flow_stop_size	integer - queue's flow stop threshold value in bytes

Key	Value
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 1.21. 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

1.9. 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.

1.9.1. 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 1.22. 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

1.9.2. Interop table by AMQP specification version

Above table represented in another format.

Table 1.23. 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

1.10. Qpid Interoperability Documentation

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

1.10.1. SASL

1.10.1.1. 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 1.24. SASL Mechanism Support

Component	ANONYMOUS	CRAM-MD5	DIGEST-MD5	EXTERNAL	GSSAPI/ Kerberos	PLAIN
C++ Broker	M3[See 10.1.1, “ Standard Mechanisms ” [65]]	M3[See 10.1.1, “ Standard Mechanisms ” [65]]		[See 10.1.1, “ Standard Mechanisms ” [65]]	M3[See 10.1.1, “ Standard Mechanisms ” [65]]	M1
C++ Client	M3[See 10.1.1, “ Standard Mechanisms ” [65]]					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

1.10.1.2. Custom Mechanisms

There have been some custom mechanisms added to our implementations.

Table 1.25. 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	

1.10.1.2.1. AMQPLAIN

1.10.1.2.2. 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).

1.11. Using Message Groups

1.11.1. 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.

1.11.2. 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.

1.11.3. 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.

1.11.4. 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.

1.11.5. 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 1.26. 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.

Option	Description
--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 1.27. 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 1.4. 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 1.5. 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}}}}")
```

1.11.5.1. 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 1.6. Overriding the default message group identifier for the broker

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

1.12. Active-Passive Messaging Clusters

1.12.1. Overview

The High Availability (HA) module provides *active-passive, hot-standby* messaging clusters to provide fault tolerant message delivery.

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 replicated to all the backups so they are always up-to-date or "hot". Backup brokers reject client connection attempts, to enforce the requirement that clients only connect to the primary.

If the primary fails, one of the backups is promoted to take over as the new primary. Clients fail-over to the new primary automatically. If there are multiple backups, the other backups also fail-over to become backups of the new primary.

This approach relies on an external *cluster resource manager* to detect failures, choose the new primary and handle network partitions. rgmanager [https://fedorahosted.org/cluster/wiki/RGManager] is supported initially, but others may be supported in the future.

1.12.1.1. Avoiding message loss

In order to avoid message loss, the primary broker *delays acknowledgement* of messages received from clients until the message has been replicated and acknowledged by all of the back-up brokers, or has been consumed from the primary queue.

This ensures that all acknowledged messages are safe: they have either been consumed or backed up to all backup brokers. Messages that are consumed *before* they are replicated do not need to be replicated. This reduces the work load when replicating a queue with active consumers.

Clients keep *unacknowledged* messages in a buffer [1] until they are acknowledged by the primary. If the primary fails, clients will fail-over to the new primary and *re-send* all their unacknowledged messages. [2]

If the primary crashes, all the *acknowledged* messages will be available on the backup that takes over as the new primary. The *unacknowledged* messages will be re-sent by the clients. Thus no messages are lost.

Note that this means it is possible for messages to be *duplicated*. In the event of a failure it is possible for a message to received by the backup that becomes the new primary *and* re-sent by the client. The application must take steps to identify and eliminate duplicates.

When a new primary is promoted after a fail-over it is initially in "recovering" mode. In this mode, it delays acknowledgement of messages on behalf of all the backups that were connected to the previous primary. This protects those messages against a failure of the new primary until the backups have a chance to connect and catch up.

[1] You can control the maximum number of messages in the buffer by setting the client's `capacity`. For details of how to set the capacity in client code see "Using the Qpid Messaging API" in *Programming in Apache Qpid*.
[2] Clients must use "at-least-once" reliability to enable re-send of unacknowledged messages. This is the default behaviour, no options need be set to enable it. For details of client addressing options see "Using the Qpid Messaging API" in *Programming in Apache Qpid*.

Not all messages need to be replicated to the back-up brokers. If a message is consumed and acknowledged by a regular client before it has been replicated to a backup, then it doesn't need to be replicated.

HA Broker States

Stand-alone	Broker is not part of a HA cluster.
Joining	Newly started broker, not yet connected to any existing primary.
Catch-up	A backup broker that is connected to the primary and downloading existing state (queues, messages etc.)
Ready	A backup broker that is fully caught-up and ready to take over as primary.
Recovering	Newly-promoted primary, waiting for backups to connect and catch up. Clients can connect but they are stalled until the primary is active.
Active	The active primary broker with all backups connected and caught-up.

1.12.1.2. Limitations

There are a some known limitations in the current implementation. These will be fixed in future 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.
- 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.
- Federation links *to* the primary will fail over correctly. Federated links *from* the primary will be lost in fail over, they will not be re-connected to the new primary. It is possible to work around this by replacing the `qpidd-primary` start up script with a script that re-creates federation links when the primary is promoted.

1.12.2. Virtual IP Addresses

Some resource managers (including **rgmanager**) support *virtual IP addresses*. A virtual IP address is an IP address that can be relocated to any of the nodes in a cluster. The resource manager associates this address with the primary node in the cluster, and relocates it to the new primary when there is a failure. This simplifies configuration as you can publish a single IP address rather than a list.

A virtual IP address can be used by clients to connect to the primary. The following sections will explain how to configure virtual IP addresses for clients or brokers.

1.12.3. Configuring the Brokers

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

Note

Broker management is required for HA to operate, it is enabled by default. The option `mgmt-enable` must not be set to "no"

Note

Incorrect security settings are a common cause of problems when getting started, see Section 1.12.9, “Security and Access Control.”.

Table 1.28. Broker 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-queue-replication` *`yes`*`|`*`no`*	Enable replication of specific queues without joining a cluster, see Section 1.13, “Replicating Queues with the HA module”.
`ha-brokers-url` *`URL`*	The URL [a] used by cluster brokers to connect to each other. The URL should contain a comma separated list of the broker addresses, rather than a virtual IP address.
`ha-public-url` *`URL`*	This option is only needed for backwards compatibility if you have been using the `amq.failover` exchange. This exchange is now obsolete, it is recommended to use a virtual IP address instead. If set, this URL is advertised by the `amq.failover` exchange and overrides the broker option `known-hosts-url`
`ha-replicate` *`VALUE`*	Specifies whether queues and exchanges are replicated by default. *`VALUE`* is one of: `none`, `configuration`, `all`. For details see Section 1.12.7, “Controlling replication of queues and exchanges”.
`ha-username` *`USER`* `ha-password` *`PASS`* `ha-mechanism` *`MECHANISM`*	Authentication settings used by HA brokers to connect to each other, see Section 1.12.9, “Security and Access Control.”
`ha-backup-timeout`*`SECONDS`* [b]	Maximum time that a recovering primary will wait for an expected backup to connect and become ready.
`link-maintenance-interval` *`SECONDS`* [b]	HA uses federation links to connect from backup to primary. Backup brokers check the link to the primary on this interval and re-connect if need be. Default 2 seconds. Set lower for faster failover, e.g. 0.1 seconds. Setting too low will result in excessive link-checking on the backups.
`link-heartbeat-interval` *`SECONDS`* [b]	HA uses federation links to connect from backup to primary. If no heart-beat is received for twice this interval the primary will consider that backup dead (e.g. if backup is hung or partitioned.) This interval is also used to time-out for broker status checks, it may take up to this interval for rgmanager to detect a hung or partitioned broker.

Options for High Availability Messaging Cluster	
	Clients sending messages may be held up during this time. Default 120 seconds: you will probably want to set this to a lower value e.g. 10. If set too low rgmanager may consider a slow broker to have failed and kill it.

[a] The full format of the URL is given by this grammar:

```
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]'
```

[b] Values specified as *SECONDS* can be a fraction of a second, e.g. "0.1" for a tenth of a second. They can also have an explicit unit, e.g. 10s (seconds), 10ms (milliseconds), 10us (microseconds), 10ns (nanoseconds)

To configure a HA cluster you must set at least `ha-cluster` and `ha-brokers-url`.

1.12.4. The Cluster Resource Manager

Broker fail-over is managed by a *cluster resource manager*. An integration with rgmanager [https://fedorahosted.org/cluster/wiki/RGManager] is provided, but it is possible to integrate with other resource managers.

The resource manager is responsible for starting the **qpidd** broker on each node in the cluster. The resource manager then *promotes* one of the brokers to be the primary. The other brokers connect to the primary as backups, using the URL provided in the `ha-brokers-url` configuration option.

Once connected, the backup brokers synchronize their state with the primary. When a backup is synchronized, or "hot", it is ready to take over if the primary fails. Backup brokers continually receive updates from the primary in order to stay synchronized.

If the primary fails, backup brokers go into fail-over mode. The resource manager must detect the failure and promote one of the backups to be the new primary. The other backups connect to the new primary and synchronize their state with it.

The resource manager is also responsible for protecting the cluster from *split-brain* conditions resulting from a network partition. A network partition divide a cluster into two sub-groups which cannot see each other. Usually a *quorum* voting algorithm is used that disables nodes in the inquorate sub-group.

1.12.5. Configuring with rgmanager as resource manager

This section assumes that you are already familiar with setting up and configuring clustered services using **cman** and **rgmanager**. It will show you how to configure an active-passive, hot-standby **qpidd** HA cluster with **rgmanager**.

Note

Once all components are installed it is important to take the following step:

```
chkconfig rgmanager on
chkconfig cman on
```

```
chkconfig qpidd off
```

The qpidd service must be *off* in `chkconfig` because `rgmanager` will start and stop `qpidd`. If the normal system init process also attempts to start and stop qpidd it can cause rgmanager to lose track of qpidd processes. The symptom when this happens is that `clustat` shows a `qpidd` service to be stopped when in fact there is a `qpidd` process running. The `qpidd` log will show errors like this:

```
critical Unexpected error: Daemon startup failed: Cannot lock /var/lib/qpidd/lo
```

You must provide a `cluster.conf` file to configure **cman** and **rgmanager**. Here is an example `cluster.conf` file for a cluster of 3 nodes named node1, node2 and node3. We will go through the configuration step-by-step.

```
<?xml version="1.0"?>
<!--
This is an example of a cluster.conf file to run qpidd HA under rgmanager.
This example assumes a 3 node cluster, with nodes named node1, node2 and node3.

NOTE: fencing is not shown, you must configure fencing appropriately for your clus
-->

<cluster name="qpid-test" config_version="18">
  <!-- The cluster has 3 nodes. Each has a unique nodeid and one vote
       for quorum. -->
  <clusternodes>
    <clusternode name="node1.example.com" nodeid="1"/>
    <clusternode name="node2.example.com" nodeid="2"/>
    <clusternode name="node3.example.com" nodeid="3"/>
  </clusternodes>

  <!-- Resouce Manager configuration. -->

   status_poll_interval is the interval in seconds that the resource manager check
   of managed services. This affects how quickly the manager will detect failed se
   -->
  <rm status_poll_interval="1">
    <!--
 There is a failoverdomain for each node containing just that node.
 This lets us stipulate that the qpidd service should always run on each node.
    -->
    <failoverdomains>
      <failoverdomain name="node1-domain" restricted="1">
 <failoverdomainnode name="node1.example.com"/>
      </failoverdomain>
      <failoverdomain name="node2-domain" restricted="1">
 <failoverdomainnode name="node2.example.com"/>
      </failoverdomain>
      <failoverdomain name="node3-domain" restricted="1">
 <failoverdomainnode name="node3.example.com"/>
```

```
      </failoverdomain>
    </failoverdomains>

    <resources>
      <!-- This script starts a qpidd broker acting as a backup. -->
      <script file="/etc/init.d/qpidd" name="qpidd"/>

      <!-- This script promotes the qpidd broker on this node to primary. -->
      <script file="/etc/init.d/qpidd-primary" name="qpidd-primary"/>

      <!--
          This is a virtual IP address for client traffic.
   monitor_link="yes" means monitor the health of the NIC used for the VIP.
   sleeptime="0" means don't delay when failing over the VIP to a new address.
      -->
      <ip address="20.0.20.200" monitor_link="yes" sleeptime="0"/>
    </resources>

    <!-- There is a qpidd service on each node, it should be restarted if it fails
    <service name="node1-qpidd-service" domain="node1-domain" recovery="restart">
      <script ref="qpidd"/>
    </service>
    <service name="node2-qpidd-service" domain="node2-domain" recovery="restart">
      <script ref="qpidd"/>
    </service>
    <service name="node3-qpidd-service" domain="node3-domain"  recovery="restart">
      <script ref="qpidd"/>
    </service>

    <!-- There should always be a single qpidd-primary service, it can run on any
    <service name="qpidd-primary-service" autostart="1" exclusive="0" recovery="re
      <script ref="qpidd-primary"/>
      <!-- The primary has the IP addresses for brokers and clients to connect. --
      <ip ref="20.0.20.200"/>
    </service>
  </rm>
</cluster>
```

There is a `failoverdomain` for each node containing just that one node. This lets us stipulate that the qpidd service should always run on all nodes.

The `resources` section defines the **qpidd** script used to start the **qpidd** service. It also defines the **qpid-primary** script which does not actually start a new service, rather it promotes the existing **qpidd** broker to primary status.

The `resources` section also defines a virtual IP address for clients: `20.0.20.200`.

`qpidd.conf` should contain these lines:

```
ha-cluster=yes
ha-brokers-url=20.0.20.1,20.0.20.2,20.0.20.3
```

The brokers connect to each other directly via the addresses listed in **ha-brokers-url**. Note the client and broker addresses are on separate sub-nets, this is recommended but not required.

The `service` section defines 3 `qpidd` services, one for each node. Each service is in a restricted fail-over domain containing just that node, and has the `restart` recovery policy. The effect of this is that rgmanager will run **qpidd** on each node, restarting if it fails.

There is a single `qpidd-primary-service` using the **qpidd-primary** script which is not restricted to a domain and has the `relocate` recovery policy. This means rgmanager will start **qpidd-primary** on one of the nodes when the cluster starts and will relocate it to another node if the original node fails. Running the `qpidd-primary` script does not start a new broker process, it promotes the existing broker to become the primary.

1.12.5.1. Shutting down qpidd on a HA node

As explained above both the per-node `qpidd` service and the re-locatable `qpidd-primary` service are implemented by the same `qpidd` daemon.

As a result, stopping the `qpidd` service will not stop a `qpidd` daemon that is acting as primary, and stopping the `qpidd-primary` service will not stop a `qpidd` process that is acting as backup.

To shut down a node that is acting as primary you need to shut down the `qpidd` service *and* relocate the primary:

```
clusvcadm -d somenode-qpidd-service
clusvcadm -r qpidd-primary-service
```

This will shut down the `qpidd` daemon on that node and prevent the primary service service from relocating back to the node because the qpidd service is no longer running there.

1.12.6. Broker Administration Tools

Normally, clients are not allowed to connect to a backup broker. However management tools are allowed to connect to a backup brokers. If you use these tools you *must not* add or remove messages from replicated queues, nor create or delete replicated queues or exchanges as this will disrupt the replication process and may cause message loss.

qpid-ha allows you to view and change HA configuration settings.

The tools **qpid-config**, **qpid-route** and **qpid-stat** will connect to a backup if you pass the flag **ha-admin** on the command line.

1.12.7. Controlling replication of queues and exchanges

By default, queues and exchanges are not replicated automatically. You can change the default behaviour by setting the `ha-replicate` configuration option. It has one of the following values:

- *all*: Replicate everything automatically: queues, exchanges, bindings and messages.
- *configuration*: Replicate the existence of queues, exchange and bindings but don't replicate messages.
- *none*: Don't replicate anything, this is the default.

You can over-ride the default for a particular queue or exchange by passing the argument `qpid.replicate` when creating the queue or exchange. It takes the same values as `ha-replicate`

Bindings are automatically replicated if the queue and exchange being bound both have replication `all` or `configuration`, 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 exchanges via the client API, add a `node` entry to the address like this:

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

There are some built-in exchanges created automatically by the broker, these exchanges are never replicated. The built-in exchanges are the default (nameless) exchange, the AMQP standard exchanges (`amq.direct`, `amq.topic`, `amq.fanout` and `amq.match`) and the management exchanges (`qpid.management`, `qmf.default.direct` and `qmf.default.topic`)

Note that if you bind a replicated queue to one of these exchanges, the binding will *not* be replicated, so the queue will not have the binding after a fail-over.

1.12.8. Client Connection and Fail-over

Clients can only connect to the primary broker. Backup brokers reject any connection attempt by a client. Clients rejected by a backup broker will automatically fail-over until they connect to the primary.

Clients are configured with the URL for the cluster (details below for each type of client). There are two possibilities

- The URL contains multiple addresses, one for each broker in the cluster.
- The URL contains a single *virtual IP address* that is assigned to the primary broker by the resource manager. This is the recommended configuration.

In the first case, clients will repeatedly re-try each address in the URL until they successfully connect to the primary. In the second case the resource manager will assign the virtual IP address to the primary broker, so clients only need to re-try on a single address.

When the primary broker fails, clients re-try all known cluster addresses until they connect to the new primary. The client re-sends any messages that were previously sent but not acknowledged by the broker at the time of the failure. Similarly messages that have been sent by the broker, but not acknowledged by the client, are re-queued.

TCP can be 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 following sections explain how to enable heartbeat in each client.

Note: the following sections explain how to configure clients with multiple dresses, but if you are using a virtual IP address you only need to configure that one address for clients, you don't need to list all the addresses.

Suppose your cluster has 3 nodes: `node1`, `node2` and `node3` all using the default AMQP port, and you are not using a virtual IP address. To connect a client you need to specify the address(es) and set the `reconnect` property to `true`. The following sub-sections show how to connect each type of client.

1.12.8.1. C++ clients

With the C++ client, you specify multiple cluster addresses in a single URL [3] You also need to specify the connection option `reconnect` to be true. For example:

```
qpid::messaging::Connection c("node1,node2,node3","{reconnect:true}");
```

Heartbeats are disabled by default. You can enable them by specifying a heartbeat interval (in seconds) for the connection via the `heartbeat` option. For example:

```
qpid::messaging::Connection c("node1,node2,node3","{reconnect:true,heartbeat:10}")
```

1.12.8.2. Python clients

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

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

Heartbeats are disabled by default. You can enable them by specifying a heartbeat interval (in seconds) for the connection via the 'heartbeat' option. For example:

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

1.12.8.3. Java JMS Clients

In Java JMS clients, client fail-over is handled automatically if it is enabled in the connection. You can configure a connection to use fail-over using the **failover** property:

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

This property can take three values:

[3] The full grammar for the URL is:

```
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]'
```

Fail-over 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	Fail-over is not supported; the connection is to a single broker only.

In a Connection URL, heartbeat is set using the **heartbeat** 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
```

1.12.9. Security and Access Control.

This section outlines the HA specific aspects of security configuration. Please see Section 1.5, “Security” for more details on enabling authentication and setting up Access Control Lists.

Note

Unless you disable authentication with `auth=no` in your configuration, you *must* set the options below and you *must* have an ACL file with at least the entry described below.

Backups will be *unable to connect to the primary* if the security configuration is incorrect. See also Section 1.12.12.2, “Authentication and ACL failures”

When authentication is enabled you must set the credentials used by HA brokers with following options:

Table 1.29. HA Security Options

HA Security Options	
`ha-username` *`USER`*	User name for HA brokers. Note this must *not* include the `@QPID` suffix.
`ha-password` *`PASS`*	Password for HA brokers.
`ha-mechanism` *`MECHANISM`*	Mechanism for HA brokers. Any mechanism you enable for broker-to-broker communication can also be used by a client, so do not use ha-mechanism=ANONYMOUS in a secure environment.

This identity is used to authorize federation links from backup to primary. It is also used to authorize actions on the backup to replicate primary state, for example creating queues and exchanges.

When authorization is enabled you must have an Access Control List with the following rule to allow HA replication to function. Suppose `ha-username`=*`USER`*

```
acl allow USER@QPID all all
```

1.12.10. Integrating with other Cluster Resource Managers

To integrate with a different resource manager you must configure it to:

- Start a qpidd process on each node of the cluster.
- Restart qpidd if it crashes.
- Promote exactly one of the brokers to primary.
- Detect a failure and promote a new primary.

The **qpid-ha** command allows you to check if a broker is primary, and to promote a backup to primary.

To test if a broker is the primary:

```
qpid-ha -b broker-address status --expect=primary
```

This will return 0 if the broker at *broker-address* is the primary, non-0 otherwise.

To promote a broker to primary:

```
qpid-ha --cluster-manager -b broker-address promote
```

Note that `promote` is considered a "cluster manager only" command. Incorrect use of `promote` outside of the cluster manager could create a cluster with multiple primaries. Such a cluster will malfunction and lose data. "Cluster manager only" commands are not accessible in **qpid-ha** without the `--cluster-manager` option.

To list the full set of commands use:

```
qpid-ha --cluster-manager --help
```

1.12.11. Using a message store in a cluster

If you use a persistent store for your messages then each broker in a cluster will have its own store. If the entire cluster fails and is restarted, the *first* broker that becomes primary will recover from its store. All the other brokers will clear their stores and get an update from the primary to ensure consistency.

1.12.12. Troubleshooting a cluster

This section applies to clusters that are using rgmanager as the cluster manager.

1.12.12.1. No primary broker

When you initially start a HA cluster, all brokers are in `joining` mode. The brokers do not automatically select a primary, they rely on the cluster manager `rgmanager` to do so. If `rgmanager` is not running or is not configured correctly, brokers will remain in the `joining` state. See Section 1.12.5, “Configuring with **rgmanager** as resource manager”

1.12.12.2. Authentication and ACL failures

If a broker is unable to establish a connection to another broker in the cluster due to authentication or ACL problems the logs may contain errors like the following:

```
info SASL: Authentication failed: SASL(-13): user not found: Password verification
```

```
warning Client closed connection with 320: User anonymous@QPID federation connecti
```

```
warning Client closed connection with 320: ACL denied anonymous@QPID creating a fe
```

Set the HA security configuration and ACL file as described in Section 1.12.9, “Security and Access Control.”. Once the cluster is running and the primary is promoted , run:

```
qpid-ha status --all
```

to make sure that the brokers are running as one cluster.

1.12.12.3. Slow recovery times

The following configuration settings affect recovery time. The values shown are examples that give fast recovery on a lightly loaded system. You should run tests to determine if the values are appropriate for your system and load conditions.

1.12.12.3.1. cluster.conf:

```
<rm status_poll_interval=1>
```

status_poll_interval is the interval in seconds that the resource manager checks the status of managed services. This affects how quickly the manager will detect failed services.

```
<ip address="20.0.20.200" monitor_link="yes" sleeptime="0"/>
```

This is a virtual IP address for client traffic. monitor_link="yes" means monitor the health of the network interface used for the VIP. sleeptime="0" means don't delay when failing over the VIP to a new address.

1.12.12.3.2. qpidd.conf

```
link-maintenance-interval=0.1
```

Interval for backup brokers to check the link to the primary re-connect if need be. Default 2 seconds. Can be set lower for faster fail-over. Setting too low will result in excessive link-checking activity on the broker.

```
link-heartbeat-interval=5
```

Heartbeat interval for federation links. The HA cluster uses federation links between the primary and each backup. The primary can take up to twice the heartbeat interval to detect a failed backup. When a sender sends a message the primary waits for all backups to acknowledge before acknowledging to the sender. A disconnected backup may cause the primary to block senders until it is detected via heartbeat.

This interval is also used as the timeout for broker status checks by rgmanager. It may take up to this interval for rgmanager to detect a hung broker.

The default of 120 seconds is very high, you will probably want to set this to a lower value. If set too low, under network congestion or heavy load, a slow-to-respond broker may be re-started by rgmanager.

1.12.12.4. Total cluster failure

Note: for definition of broker states *joining*, *catch-up*, *ready*, *recovering* and *active* see HA Broker States

The cluster can only guarantee availability as long as there is at least one active primary broker or ready backup broker left alive. If all the brokers fail simultaneously, the cluster will fail and non-persistent data will be lost.

While there is an active primary broker, clients can get service. If the active primary fails, one of the "ready" backup brokers will take over, recover and become active. Note a backup can only be promoted to primary if it is in the "ready" state (with the exception of the first primary in a new cluster where all brokers are in the "joining" state)

Given a stable cluster of N brokers with one active primary and N-1 ready backups, the system can sustain up to N-1 failures in rapid succession. The surviving broker will be promoted to active and continue to give service.

However at this point the system *cannot* sustain a failure of the surviving broker until at least one of the other brokers recovers, catches up and becomes a ready backup. If the surviving broker fails before that the cluster will fail in one of two modes (depending on the exact timing of failures)

1.12.12.4.1. 1. The cluster hangs

All brokers are in joining or catch-up mode. rgmanager tries to promote a new primary but cannot find any candidates and so gives up. clustat will show that the qpidd services are running but the the qpidd-primary service has stopped, something like this:

```
Service Name                    Owner (Last)                    State
------- ----                    ----- ------                    -----
service:mrg33-qpidd-service     20.0.10.33                      started
service:mrg34-qpidd-service     20.0.10.34                      started
service:mrg35-qpidd-service     20.0.10.35                      started
service:qpidd-primary-service   (20.0.10.33)                    stopped
```

Eventually all brokers become stuck in "joining" mode, as shown by: `qpid-ha status --all`

At this point you need to restart the cluster in one of the following ways:

1. Restart the entire cluster: In `luci:`*`your-cluster`*`:Nodes` click reboot to restart the entire cluster

2. Stop and restart the cluster with `ccs --stopall; ccs --startall`
3. Restart just the Qpid services:In `luci:`*`your-cluster`*`:Service Groups`
   a. Select all the qpidd (not qpidd-primary) services, click restart
   b. Select the qpidd-primary service, click restart
4. Stop the `qpidd-primary` and `qpidd` services with `clusvcadm`, then restart (qpidd-primary last)

1.12.12.4.2. 2. The cluster reboots

A new primary is promoted and the cluster is functional but all non-persistent data from before the failure is lost.

1.12.12.5. Fencing and network partitions

A network partition is a a network failure that divides the cluster into two or more sub-clusters, where each broker can communicate with brokers in its own sub-cluster but not with brokers in other sub-clusters. This condition is also referred to as a "split brain".

Nodes in one sub-cluster can't tell whether nodes in other sub-clusters are dead or are still running but disconnected. We cannot allow each sub-cluster to independently declare its own qpidd primary and start serving clients, as the cluster will become inconsistent. We must ensure only one sub-cluster continues to provide service.

A *quorum* determines which sub-cluster continues to operate, and *power fencing* ensures that nodes in non-quorate sub-clusters cannot attempt to provide service inconsistently. For more information see:

https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/High_Availability_Add-On_Overview/index.html, chapter 2. Quorum and 4. Fencing.

1.13. Replicating Queues with the HA module

As well as support for an active-passive cluster, the HA module allows you to replicate individual queues, even if the brokers are not in a cluster. The *original* queue is used as normal. The *replica* queue is updated automatically as messages are added to or removed from the original queue.

> **Warning**
>
> 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 replica is not used until it has been disconnected from the original.

1.13.1. Replicating queues

To create a replica queue, the HA module must be loaded on both the original and replica brokers (it is loaded by default.) You also need to set the configuration option:

```
ha-queue-replication=yes
```

to enable this feature on a stand-alone broker. It is automatically enabled for brokers that are part of a cluster.

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
```

1.13.2. Replicating queues between clusters

You can replicate queues between two standalone brokers, between a standalone broker and a cluster, or between two clusters (see Section 1.12, “Active-Passive Messaging Clusters”.) For failover in a cluster there are two cases to consider.

1. When the *original* queue is on the active node of a cluster, failover is automatic. If the active node fails, the replication link will automatically reconnect and the replica will continue to be updated from the new primary.

2. When the *replica* queue is on the active node of a cluster, there is no automatic failover. However you can use the following workaround.

1.13.2.1. Work around for fail-over of replica queue in a cluster

When a primary broker fails the cluster resource manager calls a script to promote a backup broker to be the new primary. By default this script is `/etc/init.d/qpidd-primary` but you can modify that in your `cluster.conf` file (see Section 1.12.5, “Configuring with **rgmanager** as resource manager”.)

You can modify this script (on each host in your cluster) by adding commands to create your replica queues just before the broker is promoted, as indicated in the following exceprt from the script:

```
start() {
    service qpidd start
    echo -n $"Promoting qpid daemon to cluster primary: "
    ################################
    #### Add your commands here ####
    ################################
    $QPID_HA -b localhost:$QPID_PORT promote
    [ "$?" -eq 0 ] && success || failure
}
```

Your commands will be run, and your replicas created, whenever the system fails over to a new primary.

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
    --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

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

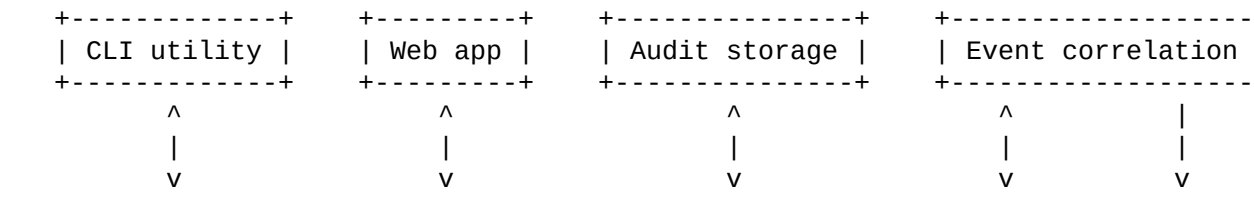
This section introduces important concepts underlying QMF.

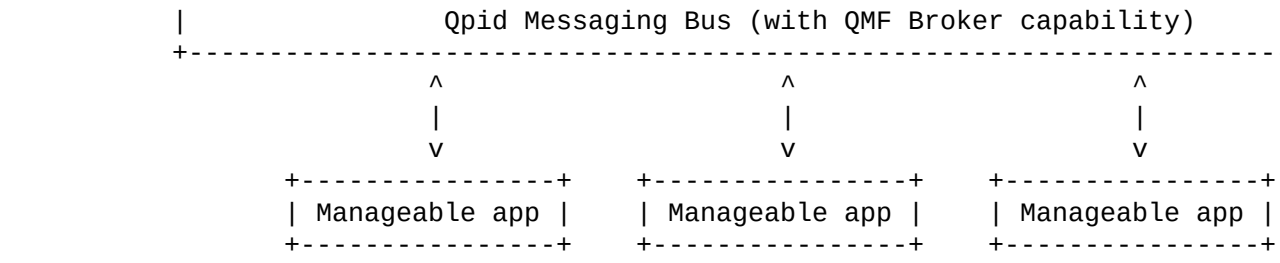
2.2.3.1. Console, Agent, and Broker

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

- Section 2.3.1, " Prerequisite - Install Qpid Messaging "
- Section 2.3.2, " Synchronous Console Operations "
- • Section 2.3.2.1, " Creating a QMF Console Session and Attaching to a Broker "
  - Section 2.3.2.2, " Accessing Managed Objects "
  - • Section 2.3.2.2.1, " Viewing Properties and Statistics of an Object "
    - Section 2.3.2.2.2, " Invoking Methods on an Object "
- Section 2.3.3, " Asynchronous Console Operations "
- • Section 2.3.3.1, " Creating a Console Class to Receive Asynchronous Data "
  - Section 2.3.3.2, " Receiving Events "
  - Section 2.3.3.3, " Receiving Objects "
  - Section 2.3.3.4, " Asynchronous Method Calls and Method Timeouts "
- Section 2.3.4, " Discovering what Kinds of Objects are Available "

2.3.1. Prerequisite - Install Qpid Messaging

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

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

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

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

2.3.2. Synchronous Console Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Connection Failed
>>>
```

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

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

2.3.2.2. Accessing Managed Objects

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

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

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

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

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

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

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

2.3.2.2.1. Viewing Properties and Statistics of an Object

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

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

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

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

```
>>> props = queue.getProperties()
>>> for prop in props:
...   print prop
```

```
...
(vhostRef, 0-0-1-0-1152921504606846979)
(name, u'reply-localhost.localdomain.32004')
(durable, False)
(autoDelete, True)
(exclusive, True)
(arguments, {})
>>>
```

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

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

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

Statistics are accessed in the same way:

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

```
>>>
```

or alternatively:

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

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

```
>>> queue.update()
>>> queue.byteTotalEnqueues
19783
>>>
```

Be Advised

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

2.3.2.2.2. Invoking Methods on an Object

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

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

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

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

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

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

```
>>> br = sess.getObjects(_class="broker", _package="org.apache.qpid.broker")[0]
>>> mlist = br.getMethods()
>>> for m in mlist:
...   print m
...
```

```
echo(sequence, body)
connect(host, port, durable, authMechanism, username, password, transport)
queueMoveMessages(srcQueue, destQueue, qty)
>>>
```

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

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

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

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

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

2.3.3. Asynchronous Console Operations

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

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

Note that console applications may always use the synchronous mechanisms.

2.3.3.1. Creating a Console Class to Receive Asynchronous Data

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

Table 2.4. QMF Python Console Class Methods

Method	Arguments	Invoked when...
brokerConnected	broker	a connection to a broker is established
brokerDisconnected	broker	a connection to a broker is lost
newPackage	name	a new package is seen on the QMF bus
newClass	kind, classKey	a new class (event or object) is seen on the QMF bus
newAgent	agent	a new agent appears on the QMF bus
delAgent	agent	an agent disconnects from the QMF bus
objectProps	broker, object	the properties of an object are published
objectStats	broker, object	the statistics of an object are published
event	broker, event	an event is published
heartbeat	agent, timestamp	a heartbeat is published by an agent
brokerInfo	broker	information about a connected broker is available to be queried
methodResponse	broker, seq, response	the result of an asynchronous method call is received

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

2.3.3.2. Receiving Events

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

Begin by importing the necessary classes:

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

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

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

```
...     print "event:", event
...
>>>
```

Make an instance of the new class:

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

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

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

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

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

2.3.3.3. Receiving Objects

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

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

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

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

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

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

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

    # If this object has not been seen before, create a new mapping from objectID
    oid = record.getObjectId()
```

```
    if oid not in queueMap:
      queueMap[oid] = record.name

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.3.4. Asynchronous Method Calls and Method Timeouts

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

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

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

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

2.3.4. Discovering what Kinds of Objects are Available