QMF Python Console Tutorial

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 Download for 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.

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.

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.

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-localhost.localdomain.32004
org.apache.qpid.broker:queue[0-1537-1-0-61] 0-0-1-0-1152921504606846979:topic-localhost.localdomain.32004
>>>
```

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.

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.

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.

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.

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.

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, rcvHeartbeats=False)
>>> 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:5672 ...
```

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 to name
    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, record.msgTotalDequeues)

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

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

# 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 to name
    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 from the map.
    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 allow
# this program to choose which objects classes it is interested in.
sess = Session(MyConsole(), manageConnections=True, rcvEvents=False, userBindings=True)

# 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.

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 *Conso le* 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.

Discovering what Kinds of Objects are Available