The AMQ Console
Operational Control for AMQ Servers

version 1.0d0

iMatix Corporation

Revised: 2006/07/18

Contents

1 Cover

1.1 State of this Document

This document is an end-user reference aimed at software developers.

1.2 Copyright and License

1.3 Abstract

We define a standard for remote administration of OpenAMQ servers; the standard consists of an XML language that defines objects and methods, and a transport mechanism that is itself built on top of AMQP.

2 Overview

2.1 Goals

The AMQ console is designed to allow remote configuration, control and management of an AMQ server, including:

- Server and virtual host configuration.
- Security configuration (authentication and access controls).
- Monitoring of key entities (queues, etc.)
- Collection of status information (performance, statistics)
- Setting of watches (on queue size, etc.)
- Administrative actions (purge queue, disconnect connection, etc.)

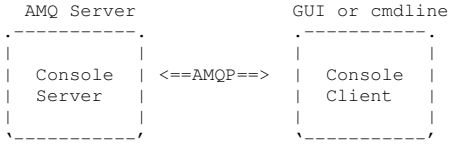
We want the console to have these features:

- To be rapid, even when many people are operating (different parts of) an AMQ server. The speed of reaction of the console is a very important part of its functionality.
- To be portable to any desired user interface technology, including web, command-line, JMX, etc. Ideally, end-users should be able to make customised operator interfaces, scripts, etc. We want the console to be user-extensible.
- To be generally compatible with conventional console abstractions, e.g. Java management beans.
- To provide access to all "operable" internals of the server, i.e. to any object that may be tuned, configured, or managed.
- To hide all aspects of the server that are not "operable", i.e. any object that should not be touched or seen. The operator must not be able to cause harm by misuse of the console.
- To have no significant impact on server performance or stability even when heavily used. The console must be non-intrusive, and wholly safe.
- Eventually, to be transactional, so that operations are completed fully, or not at all.
- Lastly, to be self-describing, using data schemas that clients can refer to, adapting statically or dynamically as suitable.

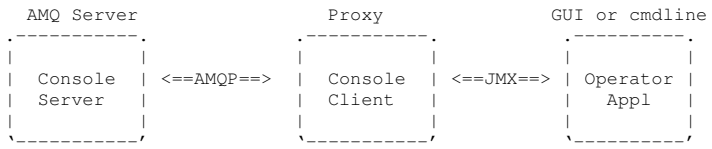
2.2 General Console Architecture

2.2.1 Server and Client Layers

The console is built using the AMQP protocol. That is, the server does not export any user interface, but rather an API on top of which arbitrary user interfaces can be built. The simplest architecture is this:



Where the console client is responsible for the user interface - command line or GUI. A more sophisticated architecture uses a console standard such as JMX so that it is easy to write arbitrary operator applications:



- The console consists of a server component and a client component.
- The console server is an application that is embedded into the AMQ server.
- The console client is an application written in an arbitrary language.
- The console client and server communicate using AMQP.

2.3 The Server Schema

We specify formally that:

- The console works on a set of entity definitions we call "classes", and a set of live entities we call "objects".
- The collection of classes that the server supports is its "schema".
- Schemas are given globally-unique names and versions.

Every AMQ server can implement its own schema, depending on its specific functionality. We do not assume that all console clients know all schemas in advance.

This is an example of a schema structure:

```
server
 :
 :-- vhost
 :    :
 :    :-- exchange
 :    :
 :    `-- queue
 :        :
 :        `-- consumer
 :
 `-- connection
```

2.4 Classes, Fields, and Methods

The schema is an arbitrarily-complex hierarchical structure representing the classes of entities in the server.

An object class is defined as follows:

- The class has a name, e.g. "server", unique within the schema.
- Each object instance has a numeric identifier.
- The class is composed of zero or more data fields (the object's properties).
- The class may allow one or more methods.

A data field is defined as follows:

- The field has a name.

- The field has a data type.
- The field may be restricted to a set of named values.
- The field has a label, used to construct a user interface.
- The field has a description, used to construct a user interface.
- The field is marked as readable and/or writeable.
- The field may contain one or more child fields.

Methods accept a number of arguments and perform some action on an object, and return a status value. A method is defined as follows:

- The method has an internal name.
- The method has a label, used to construct a user interface.
- The method carries zero or more data fields (the method's arguments).
- The method has a description, used to construct a user interface.

2.5 Commands

The API consists of these commands:

- "schema-request" - requests the server schema.
- "schema-reply" - returns the server schema.
- "inspect-request" - requests the properties of a specific object.
- "inspect-reply" - returns the properties of a specific object.
- "modify-request" - requests changes to a specific object.
- "modify-reply" - confirms changes to a specific object.
- "method-request" - executes a method on a specific object.
- "method-reply" - returns result of a method-request.
- "watch-request" - sets watches on fields of a specific object.
- "watch-reply" - returns result of a watch-request.
- "notify" - (from server to client) send a notification.

2.6 Object Identification

Each object has a unique 64-bit identifier assigned by the server when it creates the object. This identifier is used in all messages when referring to an object.

ID 0 (zero) always represents the root object.

2.7 Watches

Watch are tools that let an operator monitor specific objects without having to watch them constantly. Watches are set on specific properties of specific objects. I.e. to watch the size of two different queues requires two watches.

While the specifics of watch implementation depend on the server, we assume a model like this:

- A watch specifies an upper or lower limit for a numeric property.
- The server regularly check all watches, for instance every 10 seconds.
- When a watch limit is crossed, the console sends a notification using the AMQP topic exchange mechanisms.
- All correctly-subscribed clients can receive these notifications.

Watches have the same lifespan as the objects they are applied to. That is, durable objects have durable watches. We assume that any mechanism in the server for serialising objects is also capable of serialising watches on objects. Note that in the current OpenAMQ server implementation, all objects are temporary, and so all watches are also temporary.

3 The Console API

3.1 General Design

The console API consists of two pieces:

- Execute a command on a given object. This consists of a request from the client to the server, and a reply back from the server to the client.
- Get a notification from the server. Notifications are sent asynchronously using the AMQP topic routing mechanisms.

The console API consist of Basic content messages, exchanged using these AMQP methods:

- Basic.Publish
- Basic.Consume and Basic.Deliver
- Basic.Browse and Basic.Browse-Ok / Browse-Empty.

By using the Basic content class we ensure that the console works with all future AMQ servers. The Basic class is the only class that all AMQ servers MUST support, according to the AMQP specifications.

The request-reply mechanism uses standard queues. The client creates a temporary, private queue and binds this to amq.direct using the queue name as routing key. It then sends CML commands to the amq.system exchange, specifying "amq.console" as the routing key, and its queue name as the reply-to field. To match up responses with requests, it can set the message-id in requests - the Console will use the same value in responses.

Every request generates a response, on the condition that the reply-to field correctly refers to a queue.

We can show the client's request-reply logic using PAL scripting:

- The client creates a temporary, private reply queue and binds this to the amq.direct exchange:

```
<queue_declare queue = "" />
<queue_bind
    queue = "$queue"
    exchange = "amq.direct"
    routing_key = "$queue" />
```

- The client formats a request (in this case we read a test CML message from a text file):

```
<basic_content
    read = "console.txt"
    reply_to = "$queue"
    message_id = "id-001" />
```

- The client publishes the request to the amq.system exchange:

```
<basic_publish
    exchange = "amq.system"
    routing_key = "amq.console" />
```

- The client waits for a response on its queue:

```
<repeat>
    <basic_browse queue = "$queue" />
    <basic_returned>
        <echo>Message was returned: $message_id</echo>
        <break/>
    </basic_returned>
    <basic_arrived>
```

```
        <echo>Response arrived from console</echo>
        <break/>
    </basic_arrived>
    <wait timeout = "100" />
</repeat>
```

The notification mechanism uses topic routing and subscription queues. The client creates a temporary, private queue and binds this to the amq.notify exchange (a topic exchange), with a routing key that specifies which notifications it wants to receive. The client can bind many times with different routing keys:

We can show the client's notification logic using PAL scripting:

- The client creates a temporary, private reply queue and binds this to the amq.notify exchange:

```
<queue_declare queue = "" />
<queue_bind
    queue = "$queue"
    exchange = "amq.notify"
    routing_key = "amq.queue.*.size" />
```

- The client defines a consumer on the queue:

```
<basic_consume queue = "$queue" />
```

- The client waits for notifications on its queue:

```
<repeat>
    <wait/>
    <basic_arrived>
        <echo>Response arrived from console</echo>
    </basic_arrived>
</repeat>
```

3.2 Console Message Language

All messages between the console client and the server are formatted using an XML language we call "CML", the Console Message Language. CML uses a minimalistic XML syntax - no stylesheets, namespaces, document types, etc.

Here are some examples of CML:

```
<cml version = "1.0">
    <schema-request />
</cml>
<cml version = "1.0">
    <inspect-request object = "0" />
</cml>
<cml version = "1.0">
    <modify-request object = "0">
        <field name = "active">0</field>
    </modify-request>
</cml>
```

CML has a grammar that covers all possible server schemas. This is the overall grammar for CML messages:

```
cml             = <cml version = "1.0">
                      client-message | server-message
                  </cml>
```

This is the grammar for client messages:

```
cml             = schema-request | inspect-request | modify-request
                | method-request | watch-request
schema-request  = <schema-request />
inspect-request = <inspect-request object = "object-id" />
object-id       = 0..maxlongint-1
modify-request  = <modify-request object = "object-id">
                    [ field-data ] ...
                  </modify-request>
field-data      = <field name = "field-name">field-value</field>
field-name      = valid-name
valid-name      = {A-Za-z0-9-_.}...
field-value     = string representation of field value
method-request  = <method-request object = "object-id" name = "method-name">
                    [ field-data ] ...
                  </method-request>
method-name     = valid-name
watch-request   = <watch-request object = "object-id" [reset = "ask-reset"] >
                   [ watch-field ]...
                  </watch-request>
ask-reset       = (0) | 1
watch-field     = <field
                     name  = "field-name"
                   [ test  = "watch-type" ]
                     value = "watch-value"
                   [ route = "routing-key" ]>
                   [ alert-message ]
                  </field>
watch-type      = eq | ne | lt | le | gt | (ge)
watch-value     = integer
alert-message   = valid-text
valid-text      = {printable}...
routing-key     = valid-name
```

This is the grammar for server messages:

```
cml-command     = schema-reply | inspect-reply | modify-reply
                | method-reply | watch-reply | invalid | notify
schema-reply    = <schema-reply name = "schema-name" version = "schema-version"
                     status = "reply-status" >
                    [ schema-class ]...
                  </schema-reply>
schema-name     = valid-name
schame-version  = valid-name
reply-status    = ok | notfound | noaccess | invalid
schema-class    = <class name = "class-name" label = "class-label">
                    [ schema-field ]...
                    [ schema-method ]...
                  </class>
class-name      = valid-name
class-label     = valid-text
schema-field    = group-field | simple-field
group-field     = <group name = "field-name"
                    [ inspect = "inspect-flag" ] [ modify = "modify-flag" ]
                    [ label = "field-label" ] >
                    [ field-text ]
                    [ schema-field ]...
                  </group>
inspect-flag    = 0 | (1)
modify-flag     = (0) | 1
field-label     = valid-text
field-text      = valid-text
simple-field    = <field name = "field-name"
                    [ type = "field-type" ] [ repeat = "repeat-flag" ]
                    [ inspect = "inspect-flag" ] [ modify = "modify-flag" ]
                    [ label = "field-label" ] >
                    [ field-text ]
                    [ enum-value ] ...
                 </field>
field-type      = string | int | bool | time | objref
repeat-flag     = (0) | 1
enum-value      = <value name = "value-name">field-value</value>
value-name      = valid-text
schema-method   = <method name = "method-name" label = "method-label">
                    [ schema-field ]...
                  </method>
method-label    = valid-text
inspect-reply   = <inspect-reply class = "class-name" object = "object-id"
                    [ watches  = "watch-count" ]
                      status   = "reply-status"
                    [ notice = "notice-text" ] >
                     [ field-data ]...
                  </inspect-reply>
watch-count     = 0..maxint - 1
modify-reply    = <modify-reply
                       class = "class-name"
                       object = "object-id"
                       status = "reply-status"
                    [ notice = "notice-text" ] />
notice          = valid-text
method-reply    = <method-reply
                       class = "class-name"
                       object = "object-id"
                       status = "reply-status"
                    [ notice = "notice-text" ] />
watch-reply     = <watch-reply
                       class = "class-name"
                       object = "object-id"
                       status = "reply-status"
                    [ notice = "notice-text" ] >
                    [ watch-field ]...
                  </watch-reply>
invalid         = <invalid />
notify          = <notify object = "object-id">
                    [ notification ]...
                    [ <field name = "field-name">
                         notification
                      </field> ]...
                  </notify>
notification    = <alert>alert-message</alert>
                  <cause>cause-message</cause>
cause-message   = valid-text
```

3.3 Field Types

The goal of defining fields is to allow the automatic production of user interface forms without too much work. We allow these types of field:

- string: defines a string field. The maximum value of a string field is 255 octets when an object is modified, and no limit on output. This is the default type.
- int: defines an integer. Integers are unsigned 32 bits.
- bool: defines a boolean value, which can be shown as a check box.
- time: defines a date/time, held as a UTC time_t format.
- object: defines a relationship to another object.

String and integer fields can be enumerated, with one or more values. These would show as radio options (for 3-4 or fewer fields) or select boxes (for 5 or more options), e.g.:

```
<field name = "status" type = "int" label = "Operational status">
    <value name = "not ready"    >0</value>
    <value name = "active"       >1</value>
    <value name = "shutting down">2</value>
</field>
```

This is a minimalistic typing model that we will most likely refine over time, e.g. defining visible and internal size limits for strings and integers, and allowing multiline text fields.

3.4 Object Names

All objects must have a field called "name" that specifies a unique name for the object within the current set of objects of a particular type.

3.5 Command Implementations

3.5.1 The 'Schema' Command

The 'schema' commmand asks the server for the server schema. Any AMQ server implements exactly one schema. Schemas are named and versioned so that clients can either download schemas fully dynamically, or work with in-built schemas and verify that their servers are compatible.

An example of the schema command:

```
C: <cml version = "1.0">
      <schema-request />
   </cml>
S: <cml version = "1.0">
      <schema-reply name = "www.openamq.org/kernel" version = "0.1" status = "ok">
         <class name = "server">
            <field name = "ip-address" label = "Server IP address" />
            <field name = "port"       label = "Port for connections" />
            <field name = "version"    label = "Server software version" />
            <field name = "platform"   label = "Server platform" />
         </class>
         <class name = "vhost">
            ...
         </class>
         <class name = "connection">
            ...
         </class>
      </schema>
   </cml>
```

The reply-status field defines whether the command was successful or not. This field has these possible values:

- 'ok' means the command completed successfully.
- 'notfound' means the specified class (or object, for other commands) did not exist.
- 'noaccess' means the client application was not allowed to execute the command.
- 'invalid' means the command was wrongly formatted or incorrect.

3.5.2 The 'Inspect' Command

The 'inspect' command asks the server for a specific object. The command takes an object ID as argument, and returns all readable fields for the object, with optionally a list of all child objects.

The top-level object always has an ID zero, so the client will inspect this object first, in order to navigate the object tree.

For example:

```
C: <cml version = "1.0">
      <inspect-request object = "0" />
   </cml>
S: <cml version = "1.0">
      <inspect-reply class = "broker" object = "0" status = "ok">
         <field name = "ip-address">192.168.55.142</field>
         <field name = "port">5672</field>
         <field name = "version">0.9c2</field>
         <field name = "platform">Linux</field>
         <field name = "exchange" id = "1" />
         <field name = "exchange" id = "2" />
      </inspect-reply>
   </cml>
```

3.5.3 The 'Modify' Command

The 'modify' command asks the server to set certain fields of an object. These must be fields with the "modify" attribute set to 1 in the schema. For example:

```
C: <cml version = "1.0">
      <modify-request object = "2">
         <field name = "enabled">0</field>
      </modify-request>
   </cml>
S: <cml version = "1.0">
      <modify-reply object = "2" status = "ok" />
   </cml>
```

If the client specifies fields that are not valid (not in the modify view), the server will respond with an "invalid" status.

3.5.4 The 'Method' Command

The 'method' command provides a set of fields and a method name and asks the server to execute that method. Methods do not return values except a status code. For example:

```
C: <cml version = "1.0">
      <method-request name = "purge" object = "32" />
   </cml>
S: <cml version = "1.0">
      <method-reply name = "purge" object = "32" status = "ok" />
   </cml>
```

If the client specifies fields that are not valid (not in the modify view), the server will respond with an "invalid" status.

3.5.5 The 'Watch' Command

The 'watch' command sets, removes, or queries the watches for a specific object. The command works as follows:

- If the 'reset' option is set to 1, all existing watches on the object are deleted.
- Any watches specified are applied to the object.
- The server responds with the set of all watches on the object.

This gives the client full control over the set of watches that are active for any given object.

For example, to reset all watches:

```
C: <cml version = "1.0">
      <watch-request object = "2" reset = "1" />
   </cml>
S: <cml version = "1.0">
      <watch-reply object = "2" status = "ok" />
   </cml>
```

To add a watch to the object:

```
C: <cml version = "1.0">
      <watch-request object = "2">
         <field name = "binding-count" value = "10000">
         Exchange is getting overloaded
         </field>
      </watch-request>
   </cml>
S: <cml version = "1.0">
      <watch-request object = "2" status = "ok">
         <field name = "binding-count" test = "gt" value = "10000"
            route = "amq.exchange.2.binding-count">Exchange is
            getting overloaded
         </field>
      </watch-request>
   </cml>
```

- The watch tests are: eq, ne, lt, gt, le, ge, which correspond to the standard set of comparison operators. The default test is "ge".
- The watched value is a numeric value - only numeric and Boolean fields can be watched.
- Notifications sent to the amq.notify exchange using the routing key, which defaults to "amq.<classname>.<idvalue>.<fi

3.5.6 The 'Notify' Command

The server broadcasts warnings, error reports, and watch alarms using 'notify' commands. Any correctly subscribed AMQP client can get these notifications.

The notification consists of an alert message and an explanation (the 'cause'). The notification is attached either to an object or to a specific field in the object. For example:

```
S: <cml version = "1.0">
      <notify object = "2" >
         <field name = "binding-count">
            <alert>Exchange is getting overloaded</alert>
            <cause>Exchange 'binding-count' is now at 10200 (limit 10000)</cause>
         </field>
      </notify>
   </cml>
```

4 Worked Examples

This section must still be completed.