AMQ RFC015
AMQ Client Component API

version 0.3

iMatix Corporation <amq@imatix.com>

Revised: 2005/02/10

Contents

1 Cover

1.1 State of this Document

This document is a request for comments. Distribution of this document is currently limited to iMatix and JPMorgan internal use.

This document is unfinished. This document is a provisional proposal.

1.2 Copyright Notice

1.3 Authors

This document was written by Pieter Hintjens <ph@imatix.com>

1.4 Abstract

We define a single platform- and language-independent standard API for AMQ client components. By standardising the API for AMQP client components we reduce the workload both for the AMQP client component developers, and for their users, who need to learn only a single API.

1.5 API Definition

The current API provides serial blocking access to multiple destinations on a single server. Channels are not visible; the component opens and uses a single channel implicitly. The application can access multiple servers or simulate multiple channels by creating multiple instances of the component.

Currently we implement only queues. The API for topics and peer services will be defined in the near future.

The component is called "amq.client". It provides these properties for the connection:

identifier (write only) Specifies the client identifier to be used for the connection. The AMQP server will not allow multiple clients to connect with the same identifier.

login (write only) Specifies the login name to be used for the connection.

password (write only) Specifies the password to be used for the connection.

reply_code (read only) Returns the last reply code from the server, where 200 is OK.

reply_text (read only) Returns the last reply text from the server.

At all times the component holds a 'current message'. The same current message is used both for receiving and sending messages. The application can inspect and modify various properties of the message:

msg_content Holds the message content in a format that is defined by the message MIME type and encoding. Note that the message content can contain arbitrary binary data and is not interpreted in any way by the client component.

msg_persistent The persistent flag for the message, as a boolean value (0 or 1).

msg_priority The priority flag of the message, as a boolean value (0 or 1).

msg_expiration The message expiration date and time, in Unix time_t format.

msg_mime_type The message MIME type as a text string.

msg_encoding The message encoding as a text string.

msg_identifier The message identifier, which can contain arbitrary binary data and is not modified by the client component in any way.

After receiving a message the client can inspect these properties:

msg_handle The handle on which the message was received, an integer.

msg_number The server-assigned message number, an integer.

msg_sender The destination from which message was received, a text string.

msg_delivered The 'delivered' property - indicates that the message is to be processed by the application.

msg_redelivered The 'redelivered' property - indicates that the message may have been previously delivered.

The client component provides these methods:

connect ([hostname[, virtualpath])] Connect to the server and virtual host specified. The hostname is an IP address or DNS name, optionally followed by ':' and a port number. The default value is 'localhost'. The virtualpath is the virtual host path required. The default value is '/'. Returns 1 on success, zero on failure.

producer (destination) Opens a pre-configured or temporary queue owned by another client for writing. Returns a queue handle on success, 0 on failure.

consumer (destination[, prefetch[, noack])] Opens a pre-configured queue for reading. Prefetch is the number of messages the server will send before waiting for an acknowledgement. The default prefetch is 1. If noack is 1 the server automatically acknowledges any message it sends to the client. This is the fastest but least robust delivery method. When noack is used, prefetch is ignored - the server will send messages as fast as it receives them. Returns a queue handle on success, 0 on failure.

temporary (destination[, prefetch [, noack])] Opens a temporary queue for reading and writing. The destination name must be specified. The temporary queue may exist and have been opened by the same client identifier, in which case it is purged. Prefetch and noack are used as for consumers. Returns a queue handle on success, 0 on failure.

msg_read ([timeout)] Wait for a message to arrive from any queue opened for reading. The timeout, in seconds, tell the client component how long to wait before abandoning and returning to the application. By default, or if zero, the timeout is indefinite. When a message arrives the application can inspect it, and may forward it to another destination. Returns 1 if a message was read, 0 if the timeout expired or the server closed the connection.

msg_reset Clears all the current message's properties to their default values, namely an empty content, mime type and encoding, and zero expiration, persistence, and priority. No return value.

msg_send (handle) Sends a message to a queue destination. The application must have opened the destination and received a handle from the component. After a msg_send the current message is reset. No return value.

msg_ack () Acknowledges all messages received and not yet acknowledged. No return value.

close ([handle)] Closes a specific destination or the connection (if no handle is specified). No return value.

debug Puts the component into some implementation-defined "debug mode".

1.6 Example

Here is an example of VBScript code using a COM+ implementation of this API:

```
Set client = Server.CreateObject ("amq.client")
client.identifier = "test client"
client.login      = "username"
client.password   = "very secret"

if client.connect ("amqtest.internal", "/test") then
    orders  = client.consumer ("/queue/orders")
    reports = client.producer ("/queue/reports")
    do while client.msg_read ()
        if client.msg_handle = orders then
            response = process_the_order client.msg_content
            client.msg_reset
            client.msg_content = response
            client.msg_send (reports)
        end if
    loop
end if
```

1.7 Implementation Guidelines

Notes for component implementors:

1. Open a single channel and assign handles sequentially as the application asks for them. There is no need to maintain an internal table of destinations.
2. The consumer method requires a HANDLE OPEN followed by a HANDLE CONSUME. The producer method requires just HANDLE OPEN.

3. The 'noack' option in the consumer method maps onto the 'unreliable' parameter of the HANDLE CONSUME command.
4. The msg_read method requires reading and handling server commands until a HANDLE NOTIFY is received. If a CHANNEL CLOSE or HANDLE CLOSE is received the component should return 0 (error).
5. We assume that on all our language platforms, 1 is handled as a TRUE value and 0 as a FALSE value. Where appropriate, APIs can return TRUE or FALSE for those methods with a 1/0 return value.

At a later stage we will expand the API to cover topics, peer services, automatic fallback to secondary servers, file transfer, message browsing, and security aspects.

1.8 Security Considerations

The AMQ protocol does not yet implement specific authentication or authorisation policies. The API describe here provides for a user login and password but with no provision for privacy or authentication, we are not proposing this API for use outside controlled networks.

2 Comments on this Document

Comments by readers; these comments may be edited, incoporated, or removed by the author(s) of the document at any time.

2.1 Date, name

No comments at present.