AMQ RFC014
AMQP Agnostic Client Design

version 0.1

iMatix Corporation <amq@imatix.com>

Copyright (c) 2004 JPMorgan

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 ready for review. This document is a provisional proposal. This document describes a work in progress.

1.2 **Copyright Notice**

1.3 **Authors**

This document was written by Martin Lucina <mato@imatix.com>.

1.4 **Abstract**

We propose a language agnostic design for an AMQP/Fast client. The intent is that client implementors will use this document as a basic design for implementing AMQP/Fast wire protocol clients in any language. The design is built in a bottom-up fasion, starting with a low level wrapper around the wire protocol, and building up progressively higher levels of abstraction.

2 Introduction

2.1 Problem Statement

AMQP/Fast is designed to support many different use cases and as such is fairly large and complex. In order for AMQ to become the preferred messaging standard, we need to make it accessible to as many application developers as possible. Hence, client implemetations need to hide both protocol complexity and messaging middleware complexity from application developers.

Deciding what levels of abstraction to provide is a complex problem. Too much abstraction and you reduce functionality. Too little requires that the application developer understand more than is neccessary to use AMQP/Fast effectively.

2.2 Basic Proposal

We propose three levels of client API:

- Level 0, which makes accessible all functionality provided by the AMQP/Fast protocol. It is in fact only a lightweight wrapper that converts AMQP/Fast commands into the destination language, and is therefore totally asynchronous and stateless.

- Level 1, which makes accessible all functionality provided by the AMQP/Fast protocol but makes it synchronous. Built over Level 0.

- Level 2, "AMQ for dummies." Provides a simple API for the most frequent use cases and processing models.

3   Design Proposal

3.1   Definitions and References

Client implemetors should read this document in conjunction with AMQRFC 006, the AMPQ/Fast Wire-Level Protocol specification.

3.2   Architecture

```
+------------------------------+
|                              |
|     Application (end user)   |
|                              |
+------------------------------+
       |                 |
       |                 |
+--Client API------------------+
|              |               |
|   Level 1    |    Level 2    |
|              |               |
+------------------------------+
|                              |
|           Level 0            |
|                              |
+------------------------------+
                |
                |
+------------------------------+
|                              |
|           Network            |
|                              |
+------------------------------+
```
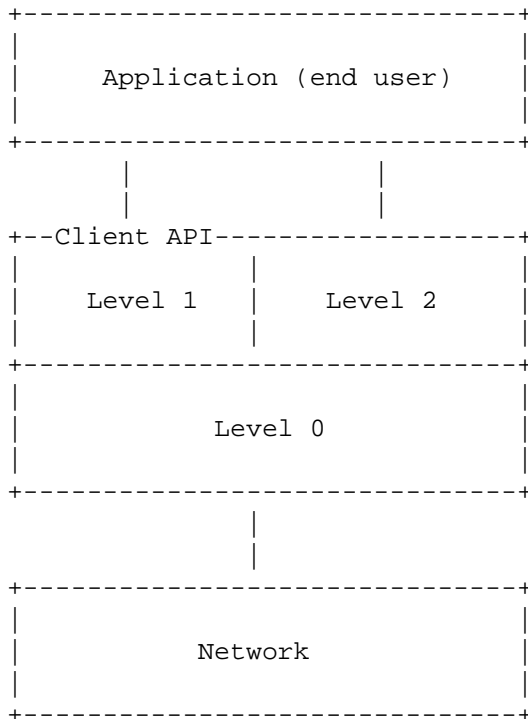
Normally, applications will be expected to use either the Level 1 or Level 2 API. [Define whether or not both can be used at the same time]. Sophisticated applications with special requirements may access the Level 0 API directly, but in this case cannot use any of the added value in the Level 1 or Level 2 APIs.

Level 1 and Level 2 APIs will use the Level 0 API internally. [And Level 2 may use Level 1, to be decided...]

3.3   Proof and Demonstration

The design will be successful if:

- Level 0 will provide complete protocol coverage

- Level 1 functionality can be built using the Level 0 API

• Level 2 functionality will cover common use cases for AMQP.

3.4 Detailed Proposal

3.4.1 Level 0 API

This level allows the user to access all functionality of AMQP/Fast. Therefore, it is just a simple wrapper around the wire protocol.

Processing is intentionally done at frame, not message level so that the client can control multiplexing of multiple channels on a single socket. This also allows the implementation to be entirely stateless, not require internal buffering of it's own and allows higher level APIs to implement multiple models for message processing (e.g. streaming vs. message-at-a-time).

While it is possible for users to use this level directly, our intent is that only a small minority of users will do so. On top of this level we will build Level 1 and Level 2 APIs.

Goals:

• Entirely stateless

• Does not impose any kind of threading model on the caller, as such is entirely asynchronous

• Implements AMQP/Fast frame serialization and deserialization

amqp_init

```
rc  = amqp_init (socket)
```

Incoming parameters:

socket Socket connected to AMQP server

Outgoing parameters:

rc Return code

Sends the two "connection initiation" bytes to the socket.

amqp_term

```
rc = amqp_term (socket)
```

Incoming parameters:

socket Socket connected to AMQP server

Outgoing parameters:

rc Return code

Dummy function, for now.

amqp_TYPE

```
rc = amqp_TYPE (socket, storage, parameters)
```

Incoming parameters:

socket Socket connected to AMQP server

storage As API is totally stateless and thus cannot have preallocated buffer, client provides a buffer to store the frame being constructed so that it wouldn't have to be allocated every time anew. Storage is supplied along with its size so that client API should be able to determine whether frame fits into buffer. If it doesn't buffer overflow error is returned.

parameters Parameters for frame type TYPE

Outgoing parameters:

rc Return code

Serialize frame parameters and send frame type TYPE to the socket.

amqp_recv (array variant)

```
(frame, rc) = amqp_recv (socket, storage)
```

Incoming parameters:

socket Socket connected to AMQP server

storage As API is totally stateless and thus cannot have preallocated buffer, client provides a buffer to store the frame being read so that it wouldn't have to be allocated every time anew. Storage is supplied along with its size so that client API should be able to determine whether frame fits into buffer. If it doesn't buffer overflow error is returned.

Outgoing parameters:

frame Frame received from socket, if any

rc Return code

Attempt to read a frame from the socket, deserialize it and return it to the caller as a structure (array of structures).

amqp_recv (callback variant)

```
rc = amqp_recv (socket, storage, callbacks)
```

Incoming parameters:

socket Socket connected to AMQP server

storage As API is totally stateless and thus cannot have preallocated buffer, client provides a buffer to store the frame being read so that it wouldn't have to be allocated every time anew. Storage is supplied along with its size so that client API should be able to determine whether frame fits into buffer. If it doesn't buffer overflow error is returned.

callbacks Array of callback functions for a individual frame types. Depending on the implementation language callbacks can be either registered in advance or passed as a parameters/closures.

Outgoing parameters:

rc Return code

Side effects:

Calls callback function depending on frame type received.

Attempt to read a frame from the socket, deserialize it and return it to the caller via function callback.

Error codes

Level 0 API defines following error codes in addition to socket errors:

AMQ_FRAME_CORRUPTED Fired when frame received from server doesn't comply with AMQP specification

AMQ_BUFFER_OVERFLOW Fired either when received frame is too big to fit into the supplied buffer, or when frame to be sent is too big to fit into the supplied buffer.

There should exist some mechanism to convert error codes into human readable strings. This could be either mechanism native to the language or a function with a following prototype:

message = amqp_strerror (errorcode)

It should be able to convert socket error codes as well as level 0 errors. May be joined with error handling mechanisms in levels 1 and 2. To be decided.

3.4.2   Level 1 API

3.4.3   Level 2 API

3.5   Alternatives

We do not propose any alternatives at this moment.

3.6   Security Considerations

Storage with restricted size is used in the Level 0 API to indicate the maximum frame size that should be accepted from the server. This is intended to protect the client from rogue servers that would attempt to send huge frames and thus exhaust client resources.

4 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.

4.1 Date, name

No comments at present.