AMQ RFC005
The Terminal-Driver API

version 0.1

iMatix Corporation <amq@imatix.com>

Revised: 2005/03/03

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

1.2 Copyright Notice

1.3 Authors

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

1.4 Abstract

OpenAMQ is based around a system of modules as described in AMQ_RFC004. Two of the main module types are terminals and drivers, which handle the main flow of messages between external clients and internal destinations. This document describes the way that terminals and drivers communicate with each other to exchange messages.

2 Introduction

2.1 Problem Statement

Our problem is to handle messages of an arbitrary size efficiently. Our real-life scenarious cover messages from a few hundred bytes to messages of many gigabytes. We need to handle multiple communication channels, one of the requirements of AMQP, and we have to be able to handle messages that are split into numerous fragments. Message confirmations are optional so that clients can send data in ”batches”, receiving a confirmation when the whole batch has been received. But, apart from these requirements, the API between terminals and drivers is a simple matter of exchanging messages and requests.

2.2 Argumentation

The terminal is an SMT-multithreaded agent that handles virtual connections called ”channels”. Some terminals handle one channel per socket, but we can have dozens or hundreds of channels per socket.

The AMQP protocol is designed to operate over this model. It is built around these dialogues (C is the client, T is the terminal):

```
C: open channel
T: channel opened

C: post full message and ask for confirmation
T: message received and stored

C: post fragment of a message without confirmation
T: (no answer)

C: post last fragment and ask for confirmation
T: message received and stored

C: request new messages
T: (no answer)

T: post new message and ask for confirmation
C: message received and stored

T: post fragment of a message without confirmation
C: (no answer)

T: post last fragment and ask for confirmation
C: message received and stored

C: close channel
T: channel closed
```

All messages have a header which provides control information such as the content length, the MIME type, and so on. The header is followed by the message body. The message body may be processed at some point depending on its MIME type, but the terminal and driver do not touch it, except to pass it along the chain between client and destination.

The basic unit of processing is the ”fragment”, which is part of, or all of a message. A small message can fit into one fragment:

```
|<------Fragment------>|
[  header  ][   body   ]
```

A larger message may be split over two or more fragments:

```
|<------Fragment------>||<------Fragment------>||<------Fragment----->|
[  header  ][   body   ][   body, second part  ][   body, last part   ]
```

The size of fragments is fixed when the client and terminal negotiate their connection; all channels between a specific client and terminal will use the same channel size.

The terminal may queue incoming fragments, but it cannot, in general, try to reconstruct a full message: messages can be much larger than the memory available. There is no obvious benefit to queuing messages in the terminal rather than passing them immediately to the driver, so this is what we do.

The driver, knowing more about the nature of the messages it handles, and needing to process them in intelligent ways, must reconstruct messages from fragments. However, this problem can be ignored for now: it does not affect the interface between terminal and driver, which is based on message fragments.

2.3 Basic Proposal

We propose (and have built) the interface around the same concept of ”channels” as used between terminal and cient. We can speak of ”internal” and ”external” channels. An internal channel connects a terminal with a driver. We use this channel to exchange commands that are very close to the AMQP commands used to send messages to and from the client. Note that AMQP also has commands for opening sessions, exchanging security information and so on. These are not part of our API.

If we label the terminal ”T” and the driver ”D”, we can sketch this API:

```
T: open channel
D: channel opened

T: post full message and ask for confirmation
D: message received and stored

T: post fragment of a message without confirmation
D: (no answer)

T: post last fragment and ask for confirmation
D: message received and stored

T: request new messages
D: (no answer)
```

```
D: forward new message and ask for confirmation
T: message forwarded to client

D: forward fragment of a message without confirmation
T: (no answer)

D: forward last fragment and ask for confirmation
T: message forwarded to client

T: close channel
D: channel closed
```

Which maps cleanly to the dialogue between client and terminal.

3 Design Proposal

3.1 Definitions and References

See AMQ_RFC008 for terminology.

3.2 Objectives

Our goals cover performance:

1. To avoid the need to queuing or copy message data.
2. To keep the overhead of the API to a strict minimum.

And scalability:

1. To remove any arbitrary limits on the size of messages.
2. To allow large numbers of messages, channels, and clients to be active at the same time.
3. To allow large and small messages to flow through the system in a fair way (IE. that the large messages do not block the smaller ones).

And quality:

1. To build the interface using a clear and high-level model.

3.3 Architecture

The current implementation consists of these design elements:

1. An iCL class "amq_driver" that provides a set of methods that a terminal may use to talk to drivers.
2. An iCL class "amq_terminal" that provides a set of methods that a driver may use to talk back to a terminal.

Both these classes are derived from the general amq_module class which is described in AMQ_RFC004.

3.4 Proof and Demonstration

The current proposal will be considered valid when we have a working implementation.

3.5 Detailed Proposal

3.5.1 The amq_driver Class

The amq_driver class provides these methods:

lookup Given a destination path, return the driver that has made a binding to that path. Note that this method is inherited from the amq_module class.

channel_open Open a new channel to a specified destination (this corresponds to the AMQP open channel command).

channel_post Post a message fragment to an open channel (this corresponds to the AMQP post command).

channel_request Request new messages from an open channel (this corresponds to the AMQP request command).

channel_confirm Tell the driver that a channel method succeeded.

channel_failure Tell the driver that a channel method failed.

channel_close Close a channel (this corresponds to the AMQP close channel command).

There is no "new" method for drivers: terminals do not create new driver objects, they can only use "lookup" to locate them.

These methods (except lookup) are asynchronous and use the SMT method model to send data to driver threads. The methods are implemented as follows:

channel_open Creates a new driver thread, and sends a "new" event to the thread along with the method arguments. Returns the reference of the thread so that the terminal can use this for following methods.

channel_post Sends a "post" event to the driver thread along with the method arguments.

channel_request Sends a "request" event to the driver thread along with the method arguments.

channel_confirm Sends a "confirm" event to the driver thread.

channel_failure Sends a "failure" event to the driver thread along with an error code.

channel_close Sends a "destroy" event to the driver thread.

All these methods return immediately, without waiting for the driver thread to finish handling the event. The driver thread has an event queue that can store an arbitrary number of waiting methods. When it needs to confirm a method (channel_open, channel_post, channel_request, or channel_close) it uses the channel_confirm and channel_failure methods provided by the amq_terminal class.

3.5.2 The amq_terminal Class

The amq_terminal class provides these methods:

channel_forward Forward a message fragment to an open channel (I.E. back to a specific client).

channel_confirm Tell the terminal that a channel method succeeded.

chanel_failure Tell the terminal that a channel method failed.

These methods are asynchronous and use the SMT method model to send data to terminal threads. The methods are implemented as follows:

channel_forward Sends a "forward" event to the terminal thread along with the method arguments.

channel_confirm Sends a "confirm" event to the terminal thread.

channel_failure Sends a "failure" event to the terminal thread along with an error code.

All these methods return immediately, without waiting for the terminal thread to finish handling the event. The terminal thread has an event queue that can store an arbitrary number of waiting methods. When it needs to confirm a method (channel_forward being the only case) it uses the channel_confirm and channel_failure methods provided by the amq_driver class.

3.5.3 The Threading Model

Our design uses one driver thread per channel, and one terminal thread per socket. One terminal thread therefore talks to N driver threads where N is the number of channels per socket that the terminal manages.

3.5.4 Queuing and Buckets

The driver and terminal methods do not copy message data. Instead, they pass pointers to buffers held by the sending party. Eliminating copying is intended to improve performance. However, it has an effect on the reuse of these buffers: while a method that refers to a particular buffer is pending, the buffer may not be used for any other work (else the method will find itself referring to a changed or invalid buffer).

We could in principle allocate a buffer when it is first needed - when the terminal accepts a message fragment from the client - and free it when it is finally processed - when the driver has stored the fragment. However we want to avoid excessive memory allocation and de-allocation.

Our solution is to build a class, amq_bucket, which manages memory buffers with a miminim of allocation/deallocation. The bucket principle is simple: the first party to fill the bucket creates a new bucket object and passes it along to other parties that need it. The last party to use a bucket destroys the object. The amq_bucket class uses a memory pool (the ipr_mempool class) to make the new and destroy operations extremely fast.

If a limit is reached on the amount of bucket memory used, threads will fall back to using fewer buckets and waiting.

3.6 Alternatives

We do not propose any alternatives at this moment.

3.7 Security Considerations

This proposal does not have any specific security considerations.

4 Comments on this Document

4.1 Date, name

No comments at present.