AMQ RFC 006
The AMQP/Fast Wire-Level Protocol

version 0.8d0

iMatix Corporation <amq@imatix.com>

Copyright (c) 2004-2005 JPMorgan

Revised: 2005/05/30

Contents

1 Cover

1.1 State of This Document

This document is a request for comments. Distribution of this document is currently limited to use by iMatix, JPMorgan, and approved partners under cover of the appropriate non-disclosure agreements.

This document is a formal standard. This document is ready for review.

1.1.1 Recent Change History

2005/05/28 - 0.8d0:

- in HANDLE CONSUME, 'dynamic' flag allows dynamic queue creation.
- in HANDLE CONSUME, 'selector' flag defines content-based selector.
- in HANDLE CONSUME, 'identifier' field removed.
- in HANDLE SEND, 'immediate' flag verifies that destination has consumers.
- in HANDLE SEND, 'selector' flag changed from string to table
- in HANDLE SEND, 'mime-type' field removed.
- in HANDLE SEND, 'streaming' field removed.
- in HANDLE NOTIFY, 'streaming' field removed.
- HANDLE CANCEL command removed.
- all reference to Peer services removed.
- all reference to streaming removed.

0.8c0

- in HANDLE CONSUME, 'unreliable' flag renamed to 'no ack' for clarity.

0.8b2

- Removed NONE security mechanism - all access requires a login, even if it is a well-known login.

0.8b1

- Temporary queues are deleted when the connection, not channel, closes.

0.8b0

- HANDLE QUERY message number requests messages GE, not GT that specified.

0.8a2

- Clarified what happens when field table contains same field twice.

0.8a1

- CONNECTION PING command has been made optional.

0.8a0

- CONNECTION TUNE command format revised to be more general.
- CONNECTION TUNE from client to server can now be sent 0+ times.
- introduced the concept of definition of field dictionaries.

1.2 Copyright Notice

This document is copyright (c) 2005 JPMorgan Inc.

1.3 Authors

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

1.4 Abstract

We propose a wire-level protocol for the AMQ open middleware architecture. The protocol - AMQP/Fast - is one of a family of AMQ protocols. AMQP/Fast is specifically aimed at managing fast multitasking connections across a reliable network. It supports standard middleware semantics and uses a binary framing model aimed at efficiency and security.

2 Introduction

2.1 Statement of Requirements

AMQ implements abstract 'destinations' held on a network of servers: its prime task to provide applications with reliable, pervasive, fast, secure and economical shared access to these destinations.

AMQP/Fast is a binary wire-level protocol that provides a general model for abstract destinations and services based on them.

2.1.1 Operational Scenarios

These are the messaging architectures we aim to support:

- Store-and-forward with N writers and 1 reader.
- Transaction distribution with N writers and N readers.
- Publish-subscribe with N writers and N readers.
- Content-based routing with N writers and N readers.
- Queued file transfer with N writers and N readers.
- Point-to-point connection between 2 peers.
- Market data distribution with N sources and N readers.

2.1.2 Design Criteria

These were the main design criteria for AMQP/Fast:

- No copying or reformatting of message data. We make a distinction between 'message data', which applications pass to each other, and other kinds of data - e.g. commands - exchanged via AMQP.
- Fully parsable with no look-ahead. This means no content parsing, as in HTTP or SMTP: all data is length-specified so the recipient can allocate the space needed before reading.
- Compatible with modern security standards. This means using SASL for authentication and privacy negotiation.
- Uses a standard model for error reporting. In other words, a standard 3-digit reply code (E.G. 200 OK).
- Uses an asynchronous, parallelised, segmented model of flow control so that latency can be reduced as required.
- Supports transaction management semantics. This means providing a way to define, start, commit, and rollback transactions.
- Supports content-type driven processing. This means providing a standard place to define the content type and encoding of all messages.

- Relatively simple to implement as a client. This means keeping the protocol as simple as possible.
- Compatible with message-oriented middleware (MOM) concepts. This means supporting, and providing clear instructions for using, standard MOM concepts like 'sessions', 'producers', and 'consumers'.
- Ability to run multiple protocols on a single server socket to support NAT firewalls.
- Negotiation of client/server capabilities. This is a standard goal for all extensible and evolving protocols.
- Inbuilt support for virtual servers managed by a single server process.
- Ability to pass message data using domain-specific out-of-band techniques like shared memory or remote-DMA.

2.2 Design Elements

2.2.1 Connection

The client and server talk across a persistent socket connection. The connection carries commands between the server and client.

We assume a reliable stream-oriented network transport layer (TCP/IP or equivalent). A server accepts connections on a well-defined or configured port and clients connect to this port. The client sends a small header to tell the server what protocol it wishes to use. The client and server then exchange commands to authenticate the client.

2.2.2 Security

Privacy and security are negotiated between server and client when the connection is opened. The security model is based on the SASL standard which allows the deployment of arbitrary security profiles (e.g. PLAIN, TLS) depending on the capabilities and requirements of client and server.

2.2.3 Virtual Hosts

The protocol supports virtual hosts: a single running server process can host an arbitrary number of independent configurations, each identified by a 'virtual access path'. The choice of virtual host is made after security negotiation, so AMQP virtual hosts share a single set of security credentials, but independent access control lists. (This design may be reviewed at a later stage if necessary.)

An AMQ virtual host is a collection of destination configurations that operate independently of other virtual hosts on the same server. The virtual access path forms part of the AMQ URL (see AMQ RFC 013).

2.2.4 Frames

Each command is formatted into a 'frame', a compact binary structure. A frame always starts with a header that tells the recipient how many octets of data follow. This header consists of either two octets holding a size 0x0 to 0xFFFE, or 0xFFFF followed by four octets holding a size 0xFFFF to 0xFFFFFFFF. In practice, the frame size is limited to under 64k octets to avoid excessive memory consumption, but all AMQP clients and servers MUST handle the frame header escaping convention as described here. Following the header are a number of 'fields', numbers and strings. Each command has a fixed set of fields, which we specify in detail in this document. Strings are formatted as a length followed by data. Some frames carry message data, others do not.

Command types are single octets as defined below in the section 'Frame Syntax'. We use mnemonic names in this document, but these names do not actually appear in the wire-level protocol.

Frame sizes are predictable, and resources needed to read frames - i.e. memory buffers - can be allocated ahead of time. If there are reasons why a particular frame cannot be processed, it can be discarded simply by skipping forward in the connection octet stream.

2.2.5 Channels and Fragmentation

For high performance communications, we reduce and eliminate latency. IETF RFC 3117 has a good discussion about this topic. To eliminate the cost of opening and closing connections, we use persistent connections. So that message data can be transferred with no copying, inspection, or transformation (all of which increase latency), we send message data using the 'octet counting' technique, i.e. we say 'here are N octets of data'. To decrease client-server latency we allow commands to be 'pipelined', so that the client can send multiple commands, each to be processed by the server in an asynchronous manner. To decrease latency in the client and server, we organise these pipelines into threads of control that match actual multithreaded processing models. To prevent starvation of any thread or monopolisation of the connection by one thread, we segment large messages into smaller fragments, sharing the connection in a fair manner.

This gives us our channel model, which defines a simple and clear way for implementors to build the asynchronous, pipelined, low-latency architecture we are aiming for.

A channel corresponds closely to a thread of control in the client and in the server. Each channel carries a sequence of commands and messages handled in a strictly serial fashion. Each channel is independent: errors on one channel do not affect other channels, and large messages sent on one channel do not starve other channels of network resources.

While there are many other plausible ways of achieving the same goal of low latency, such as connection pooling, the channel model has the advantages of being simple, standard (used in many middleware protocols and in standards such as IETF RFC3080), and entirely general.

2.2.6 Service Families

We define two principle service families: queues (also known as 'point-to-point') and topics (also known as 'publish and subscribe').

The semantics of a queue service are one of distribution. That is, the service distributes messages to one or more readers so that each message goes at most to a single reader.

The semantics of a topic service are one of subscription. That is, the service distributes messages to multiple readers depending on their previously expressed interest.

In this document, 'destination' may be used to cover queue or topic.

2.2.7 Command Hierarchy

Commands work on three levels: connection, channel, destination, as shown by this simple flow chart, where C is the client and S is the server. For brevity we do not show all confirmations:

```
#   Connection initiation and negotiation
C: initiate connection
S: CONNECTION CHALLENGE
C: CONNECTION RESPONSE
S: CONNECTION TUNE
C: CONNECTION TUNE...
#   Open connection
C: CONNECTION OPEN
    #  Channel activity
    C: CHANNEL OPEN
        #   Activity on a destination
        C: HANDLE OPEN
            S: HANDLE CREATED
            #   Sending data to a destination
            C: HANDLE SEND
            #   Getting data from a destination
            C:  HANDLE CONSUME
                S: HANDLE NOTIFY
                    C: HANDLE UNGET
            #   Browsing a destination
            C: HANDLE QUERY
                S: HANDLE INDEX
                C: HANDLE BROWSE
                    S: HANDLE NOTIFY
            #   Restartable large-message exchange
            C: HANDLE PREPARE
                S: HANDLE READY
                C: HANDLE SEND
            #   Finished with destination
            C: HANDLE CLOSE
            S: HANDLE CLOSE
            #   Stopping/restarting data
            C: HANDLE FLOW
        #   Acknowledge messages
        C: CHANNEL ACK
        #   Transaction management
        C: CHANNEL COMMIT
        C: CHANNEL ROLLBACK
        C: CHANNEL CLOSE
        S: CHANNEL CLOSE
    #   Heartbeat function
    C: CONNECTION PING
    S: CONNECTION PING
    C: CONNECTION CLOSE
    S: CONNECTION CLOSE
```

2.2.8 Command Scenarios

Although AMQP appears complex, the exchange of commands follow a small set of simple and consistent scenarios. We explain these so that the reader who is unfamiliar with AMQP is able to understand how this large protocol breaks into digestible pieces.

2.2.8.1 Preparing The Connection

The start of a connection always goes through the same set of steps.

1. The client connects to the server's port and sends the 2-octet initiation sequence.
2. The server checks this, and then replies with CONNECTION CHALLENGE.

3. The client authenticates (e.g. provides a user id and password) with a CONNECTION RESPONSE.
4. The server can loop, sending CONNECTION CHALLENGE again. This is needed in some security models that need more than one response from the client.
5. The server sends CONNECTION TUNE, a command that configures various parameters for the connection.
6. The client answers with zero or more CONNECTION TUNE commands, and then sends CONNECTION OPEN.

2.2.8.2 Opening Channels and Handles

To do any real work the client has to open at least one channel, and then at least one handle. Handles work with specific destinations.

So the dialogue looks like this:

1. The client sends CHANNEL OPEN.
2. The client sends HANDLE OPEN.
3. The client sends other handle or channel commands.

2.2.8.3 Sending Messages to The Server

The client can send messages to a destination, once it has opened a handle (and channel). It can send messages at any time - there is no restriction about how and when, except that the handle and channel must be open.

The dialogue for sending a message is simply:

1. The client sends HANDLE SEND.

2.2.8.4 Receiving Messages from The Server

To receive messages from the server the client has to explicitly say ”I’m ready to get messages”. Here the dialogue is more complex because the client can control the flow of messages in different ways.

1. The client sends HANDLE CONSUME to tell the server that it is ready to get messages from a specific destination.
2. As messages arrive, the server sends HANDLE NOTIFY commands to the client, each carrying a message or a message fragment. A message can be split into fragments: these come one after the other, with no interruption, on the channel, as HANDLE NOTIFY commands.
3. The client can pause the flow of messages by sending HANDLE FLOW.
4. The client can restart the flow of messages by sending HANDLE FLOW with a different option.
5. The client can stop all messages with HANDLE CLOSE.

2.2.8.5 Acknowledging Messages

Acknowledgment is where the client tells the server ”I’ve received the message and processed it, and you can now consider it ’delivered’.”

The client sends acknowledgements as it likes. There are several ways of doing this. It can acknowledge each message immediately when it has received it. Then we get this dialogue:

1. The server sends HANDLE NOTIFY.
2. The client collects message fragments and when it has a complete message it sends CHANNEL ACK.

Alternatively, the client can acknowledge a message when the application has actually finished working with it. This gives us a dialogue like this:

1. The server sends HANDLE NOTIFY commands.
2. The client delivers complete messages to the application.
3. At some later time, the client sends CHANNEL ACK commands to the server.

Acknowledgments are done at the channel level, not the handle level. This is because the messages on a channel are considered to be a single stream. The client says ”I’ve done up to this point in the stream”. One CHANNEL ACK acknowledges all messages up to a specified message number.

2.2.8.6 Handling Errors

Errors are handled in a fairly robust way: the server (or client, because either party can decide something is wrong) sends HANDLE CLOSE, or CHANNEL CLOSE, and shuts down the handle or channel.

Because the other party needs an opportunity to free resources, AMQP uses a simple hand-shake for error handling. Thus:

1. The server detects an error and sends CHANNEL CLOSE.
2. The client receives this command, and closes the channel and all handles in it.
3. The client then sends CHANNEL CLOSE in reply.
4. When the server receives this, it closes the channel and all handles in it.
5. At that stage, the only thing the client can do with the channel is CHANNEL OPEN.

2.2.8.7 Browsing Queues and Topics

”Browsing” means reading messages without disturbing them. You might compare this to scanning your email without actually marking the messages as ”read”.

Before a client can browse a destination it has to open a channel and handle, as usual. The dialogue for browsing is then:

1. The client sends HANDLE QUERY, asking the server for a list of all available messages in the destination.
2. The server replies with HANDLE INDEX, which is a list of the messages it has available.

3. The client sends HANDLE BROWSE to ask for a specific message.
4. The server sends back the message with a HANDLE NOTIFY. There is a flag in the HANDLE NOTIFY saying "this message is not being delivered, so don't try to process it!".
5. The client can repeat the process, sending HANDLE BROWSE as often as it needs to.

Since other clients can be reading the same destination, it's possible that a message disappears from the list, and that a HANDLE BROWSE fails. In this case the server replies with a HANDLE REPLY with an error code, instead of with a HANDLE NOTIFY. This is one of the few cases where the server tells the client explicitly that a command failed. In most cases, failure is treated as an error and the handle, channel, or connection is closed.

2.2.8.8 Confirmations

We've kept the above dialogues simplified by ignoring confirmations. In most cases, clients don't care when a command succeeds. Confirming every command creates a very "chatty" and inefficient protocol. So we tend to adopt a "fire and forget" model in which success is assumed, and failure is treated as exceptional.

For example when a client sends a message to a destination, it's very abnormal for the message to be refused. So in most cases the client can send the message and forget about it. We're using a reliable network protocol (TCP/IP) so data does not get lost.

If there is a problem - e.g. the server has run out of space - then the server closes the channel, or connection.

Having said all this, there are cases where the client wants a chatty protocol, and wants explicit confirmation of every important command. The client does this by setting the 'confirm-tag' field in the commands it wants confirmed, command by command.

The dialogue would then look like this:

1. Client sends command, with non-zero confirm-tag.
2. Server processes command and replies with CHANNEL REPLY or HANDLE REPLY (depending on the command).
3. Client sends next command.

Confirmations are never batched: if the client requests a confirmation then the server will not send any other command back to the client until the command being confirmed has completed.

When the server sends other commands (while processing the client's command), these will arrive before the CHANNEL REPLY or HANDLE REPLY command.

You can factor this into the other dialogues shown above to give the fully-confirmed dialogue for (E.G.) sending a message.

2.2.9 Command Priorities

Command priorities are trivially determined by the command hierarchy and the assumption that a single thread within the client or server is responsible for reading and dispatching commands to other threads (or to queues for later processing by the same thread). Connection commands are processed without any internal dispatching whatsoever, so have the highest priority. Channels commands are processed after dispatching to the thread responsible for the channel.

Thus a client can send a CONNECTION PING and know that the command will be handled with a higher priority than pending HANDLE SEND commands, on the same channel.

2.2.10 Error Handling

Error handling is based on this principle: so long as there are no problems processing commands, confirmations are optional and can be queued, postponed, and ignored. After the initial tuning phase, the client never needs to wait for the server to confirm a command before it sends the next command.

If a protocol/syntax problem occurs, the server shuts-down that part of the dialogue resource affected. That is, any error on a destination results in a HANDLE CLOSE. Any error on a channel results in a CHANNEL CLOSE, and any error on a connection results in a CONNECTION CLOSE.

Error reporting is done using a 3-digit code where the first digit signals the severity, the second the nature of the error, and the last provides more detail when needed.

2.2.11 Client-assigned IDs

To allow command batching (pipelining), it is the client which assigns IDs for channels, destinations, transactions, and so on. The server may check IDs and reject invalid IDs, but it never generates them. This design means the client can always open a resource (channel, destination) and then send commands using its ID, without waiting for a server reply.

2.2.12 Connection Heartbeat

Since our protocol is highly asynchronous, we cannot rely on a simple ack/nack model to know when the other party is incapacitated, for any reason including over-loading, network errors, internal failure, etc.

To allow either party to monitor and act on the status of the other we use a connection heatbeat; a small high-priority command that is sent at regular intervals. So long as the heartbeat command arrives, we know the other side is alive and kicking. If the heartbeat commands stop arriving, we can take suitable action.

For ease of implementation the heartbeats are optional: until a peer sends a first heartbeat command, it is assumed to be working without heartbeats (and the other peer can use any reasonable algorithm to decide whether the situation is normal or not.)

3 Design Proposal

3.1 Framing

3.1.1 Frame Syntax

```
frame                = initiation / command / message
initiation           = protocol-id protocol-version
protocol-id          = %d128
protocol-version     = %d1
command              = command-size command-payload command-end
command-size         = short-size / long-size
short-size           = %x0 .. %xFFFE
long-size            = %xFFFF %x0 .. %xFFFFFFFE
command-payload      = connection-challenge /
                       connection-response /
                       connection-tune /
                       connection-open /
                       connection-ping /
                       connection-reply /
                       connection-close /
                       channel-open /
                       channel-ack /
                       channel-commit /
                       channel-rollback /
                       channel-reply /
                       channel-close /
                       handle-open /
                       handle-send /
                       handle-consume /
                       handle-flow /
                       handle-unget /
                       handle-query /
                       handle-browse /
                       handle-created /
                       handle-notify /
                       handle-index /
                       handle-prepare /
                       handle-ready /
                       handle-reply /
                       handle-close /
connection-challenge  = %d10 version mechanisms challenges
version              = OCTET
mechanisms           = short-string
short-string         = OCTET *OCTET
challenges           = long-string
long-string          = short-integer *OCTET
short-integer        = 2*OCTET
connection-response  = %d11 mechanism responses
mechanism            = short-string
responses            = long-string
connection-tune      = %d12 options dictionary
options              = long-string
dictionary           = long-string
connection-open      = %d13 confirm-tag virtual-path
                       client-name options
confirm-tag          = short-integer
virtual-path         = short-string
client-name          = short-string
connection-ping      = %d14 respond
respond              = BIT
connection-reply     = %d18 confirm-tag reply-code reply-text
reply-code           = short-integer
reply-text           = short-string
connection-close     = %d19 reply-code reply-text
channel-open         = %d20 channel-id confirm-tag transacted
                       restartable options out-of-band
channel-id           = short-integer
transacted           = BIT
restartable          = BIT
```

```
out-of-band          = short-string
channel-ack          = %d21 channel-id confirm-tag message-nbr
message-nbr          = long-integer
long-integer         = 4*OCTET
channel-commit       = %d22 channel-id confirm-tag options
channel-rollback     = %d23 channel-id confirm-tag options
channel-reply        = %d28 channel-id confirm-tag reply-code
                       reply-text
channel-close        = %d29 channel-id reply-code reply-text
handle-open          = %d30 channel-id handle-id service-type
                       confirm-tag producer consumer browser
                       temporary dest-name mime-type encoding
                       options
handle-id            = short-integer
service-type         = short-integer
producer             = BIT
consumer             = BIT
browser              = BIT
temporary            = BIT
dest-name            = short-string
mime-type            = short-string
encoding             = short-string
handle-send          = %d31 handle-id confirm-tag fragment-size
                       partial out-of-band recovery dest-name
fragment-size        = long-integer
partial              = BIT
recovery             = BIT
handle-consume       = %d32 handle-id confirm-tag prefetch no-local
                       no-ack dynamic dest-name selector
prefetch             = short-integer
no-local             = BIT
no-ack               = BIT
dynamic              = BIT
handle-flow          = %d34 handle-id confirm-tag flow-pause
flow-pause           = BIT
handle-unget         = %d35 handle-id confirm-tag message-nbr
handle-query         = %d36 handle-id message-nbr dest-name selector
                       mime-type
handle-browse        = %d37 handle-id confirm-tag message-nbr
handle-created       = %d40 handle-id dest-name
handle-notify        = %d41 handle-id message-nbr fragment-size
                       partial out-of-band recovery delivered
                       redelivered dest-name
delivered            = BIT
redelivered          = BIT
handle-index         = %d42 handle-id message-nbr message-list
message-list         = long-string
handle-prepare       = %d43 handle-id fragment-size content-hash
content-hash         = short-string
handle-ready         = %d44 handle-id message-size
message-size         = long-integer
handle-reply         = %d48 handle-id confirm-tag reply-code
                       reply-text
handle-close         = %d49 handle-id reply-code reply-text
command-end          = %xCE
message              = message-head / message-fragment
message-head         = body-size persistent priority expiration
                       mime-type encoding msg-identifier headers
body-size            = long-integer
persistent           = BIT
priority             = OCTET
expiration           = long-integer
msg-identifier       = short-string
headers              = long-string
message-fragment     = *OCTET
```

Notes:

- All multi-octet integers and sizes are represented in network byte order (High-low). There is no attempt to optimise the case when two low-high systems (e.g. two Intel CPUs) talk to each other.
- Empty strings are represented by a zero length, one or two octets depending on the type of string.
- Long strings are binary-safe and may contain any octet values including binary nulls.

- Short strings are ANSI-C compatible and assumed to be in UTF-8 format. They MAY NOT contain binary zero octets.
- The command-end byte is included in the command-size.
- Message fragmentation is explained in more detail later.

3.1.2 Notes

Command structures are designed so that the variable parts (strings and string tables) are at the end. The most significant identifiers come at the start. Most strings are kept as 'short' to save space; long strings are used mainly to carry data that can have an arbitrary size. Bits are packed into octets.

3.2 Connection Commands

3.2.1 Connection Command Summary

CONNECTION INITIATION A special two-byte identification sequence that the client sends before sending or expecting any other commands.

CONNECTION CHALLENGE Server asks client to authenticate itself. Client replies with a suitable CONNECTION RESPONSE.

CONNECTION RESPONSE Client provides server with its credentials. The actual credentials depend on the security mechanisms implemented; this is not yet part of the standard proposal. Server replies with another CONNECTION CHALLENGE or a CONNECTION TUNE.

CONNECTION TUNE Server asks client to confirm tuning parameters. Client replies with zero or more CONNECTION TUNE commands, followed by a CONNECTION OPEN.

CONNECTION OPEN The client asks the server to open a connection to a virtual host.

CONNECTION PING Either party asks the other to respond with a high-priority message.

CONNECTION REPLY Server responds to a satisfactory CONNECTION OPEN with a CONNECTION REPLY to tell the client that the connection is ready.

CONNECTION CLOSE Either party closes the connection, the other confirms with CONNECTION CLOSE.

3.2.2 Connection Initiation

The client initiates the connection by sending two bytes of data:

```
protocol-id protocol-version
```

protocol-id (octet) The value 128.

protocol-version (octet) The client's highest protocol version, currently 1.

We define these protocol id numbers:

A-Z,a-z AMQP/HTTP

128 AMQP/Fast

129-160 Reserved

161-255

```
Allowed for test protocols.
```

The client and server negotiate the protocol version number as follows: the client reports the highest protocol version it can handle, the server responds with the highest version it can handle, up to the version specified by the client. If the client can handle that version, it continues, otherwise it closes the connection.

Here is an example. The client is marked as C, the server as S:

```
S: opens port 3018 to accept connections
C: connects to port
S: accepts the connection
C: 128 1
```

The server responds to an valid initiation sequence with a CONNECTION CHALLENGE command (see below) and responds to an invalid initiation sequence by closing the network connection.

3.2.3 CONNECTION CHALLENGE Command

Authentication (logging-in) is done through a series of challenges by the server and responses from the client. The server starts the process by sending a CONNECTION CHALLENGE command to the client:

```
CONNECTION CHALLENGE
    version                 octet
    mechanisms              short string
    challenges              field table
```

protocol-version The protocol version that the server agrees to use, which cannot be higher than the client's version.

mechanism-list A list of the security mechanisms that the server supports, delimited by spaces. Currently AMQP supports these mechanisms: PLAIN. See below for details.

challenges A set of challenge fields that specify the information asked for for the current (or all) mechanisms. Defined below, for each security mechanism.

The maximum allowed size for a CONNECTION CHALLENGE command frame is 4096 octets.

3.2.3.1 Field Tables

Many commands contain 'field table' structures. These are long strings that contain name-value pairs. Each name-value pair is a structure that provides a field name, a field type, and a field value. Fields can store a string, an integer, a decimal, or a date and/or time. This is a syntax chart for field tables:

```
field-table      = short-integer *field-value-pair
field-value-pair = field-name field-value
field-name       = short-string
field-value      = 'S' long-string
                 / 'I' long-integer
                 / 'D' decimal-value
                 / 'T' long-integer
decimal-value    = decimals long-integer
decimals         = OCTET
```

Notes:

- Field names must start with a letter, '$' or '#' and may continue with letters, '$' or '#', digits, or underlines, to a maximum length of 128 characters.
- String values are formatted as long strings, i.e. with a 2-octet length field in network order, followed by that number of data octets.
- Integer values are signed (I.E. 31 bits plus sign).
- Decimal values are not intended to support floating point values, but rather business values such as currency rates and amounts. The 'decimals' octet is not signed.
- Time values are held in the POSIX time_t format with an accuracy of one second.

As a convention when defining allowed values in field tables we will use this syntax:

```
Name=(Type)Content
```

For example:

```
EXPIRATION-TIME=(Time)
IDENTIFIER=(String)01-ABCD-9876
MAXIMUM-SIZE=(Integer)
```

If a field occurs more than once in the field table only the first instance should be used.

3.2.4 CONNECTION RESPONSE Command

The client responds to a CONNECTION CHALLENGE with a CONNECTION RESPONSE command:

```
CONNECTION RESPONSE
    mechanism              short string
    responses              field table
```

mechanism A single security mechanisms selected by the client; one of those specified by the server.

responses A set of response fields that provide additional information for the specific security mechanism chosen.

There are three possible server responses to a CONNECTION RESPONSE. First, the server needs more information to continue, so it sends a new CONNECTION CHALLENGE asking for this information. Second, the server detects an error in the command (syntax error, invalid user credentials, etc.), in which case it closes the connection. Last, the server accepts the CONNECTION RESPONSE and it replies with a CONNECTION TUNE command.

The maximum allowed size for a CONNECTION RESPONSE command frame is 4096 octets.

3.2.4.1 The PLAIN Security Mechanism

The PLAIN security mechanism passes login credentials as clear text, with no attempt at encryption or authentication. This mechanism is not secure in any way.

When the server sends a CONNECTION CHALLENGE specifying the security mechanism ”PLAIN”, the client should respond with a CHALLENGE RESPONSE containing a responses table with two fields, LOGIN and PASSWORD. These fields provide the login name and plain-text password:

```
LOGIN=(String)
PASSWORD=(String)
```

Note that the field names used in a field table are case-sensitive. AMQP uses uppercase in all pre-defined field names. Given a login ”Johan” and a password ”Ok”, the binary format of the field table would be:

```
%0d %28d
%5d 'L' 'O' 'G' 'I' 'N' 'S' %0d %5d 'J' 'o' 'h' 'a' 'n'
%8d 'P' 'A' 'S' 'S' 'W' 'O' 'R' 'D' 'S' %0d %2d 'O' 'k'
```

3.2.5 CONNECTION TUNE Command

After authentication the client and server negotiates various parameters for the connection. This negotiation consists of an exchange of CONNECTION TUNE commands (one from the server to the client and zero or more in response back from the client to the server):

```
CONNECTION TUNE
    options             field table
    dictionary          field table
```

options A set of tuning options as defined below.

dictionary A client-defined set of aliases for field names used in messages. This field is always empty in the command sent from server to client.

The server proposes values for the -max parameters and the client confirms with the parameters it proposes, which must be equal to or less than the values proposed by the server. If the server does not accept the client’s values it replies with CONNECTION CLOSE.

The maximum allowed size for a CONNECTION TUNE command frame is 4096 octets.

3.2.5.1 Connection Tuning Options

The server may send these fields to the client in the 'options' field table:

```
FRAME_MAX=(Integer)
CHANNEL_MAX=(Integer)
HANDLE_MAX=(Integer)
HEARTBEAT=(Integer)
TXN_LIMIT=(Integer)
```

FRAME_MAX The largest frame size or fragment size that the server is prepared to accept, in octets. May not be zero. The frame-max must be large enough for any command frame and not less than 1024.

CHANNEL_MAX The maximum total number of channels supported per connection. Zero means 'unlimited'.

HANDLE_MAX The maximum total number of handles supported per connection. Zero means 'unlimited'.

HEARTBEAT The delay, in seconds, of the connection heartbeat requested by the server: see the CONNECTION PING command. Zero means the server does not request a heartbeat.

TXN_LIMIT The highest number of messages that the server will accept per transaction. If the server does not limit the size of transactions, this field will either be specified as zero, or not specified.

The client may send these fields to the server in the 'options' field table:

```
FRAME_MAX=(Integer)
HEARTBEAT=(Integer)
```

FRAME_MAX The largest frame size or fragment size that the client is prepared to accept, in octets. May not be larger than the value sent by the server, and may not be smaller than 1024.

HEARTBEAT The delay, in seconds, of the connection heartbeat requested by the client: see the CONNECTION PING command. Zero means the client does not request a heartbeat.

The heartbeat parameter is a hint, not a formal request. That is: the receiving party may but does not have to send CONNECTION PING commands accordingly. We use this rule: if a party sends at least one CONNECTION PING, it must continue to send these commands as defined by the heartbeat request it received. If one party does not send CONNECTION PING commands at all, the other party must use another, reasonable, algorithm for monitoring the state of the connection. (E.g. a timeout after the last received command.)

3.2.5.2 Dictionaries

The 'dictionary' is a simple technique designed to save network transfer costs. Messages have a 'headers' field that holds message headers (field name-value pairs). Since the field names used are generally the same for all messages sent to a queue, we can see large amounts of repetitive data being sent across the network.

A dictionary defines short names for message header fields. Thus a field who's full name is "Expiration-Time" might have a short name "ET". When the message fields are used in selection and routing, we use the full name. However when the message is carried by a HANDLE SEND or HANDLE NOTIFY, we use the short name.

This is specified in the dictionary using the field name-value pair:

```
ET=(String)Expiration-Time
```

The client can send as many dictionaries as it wants to, each in a CONNECTION TUNE command; dictionaries are merged into a single set of aliases that apply to all messages sent across the connection.

Note	**selectors are currently unimplemented, and dictionaries are**

3.2.6 CONNECTION OPEN Command

After sending CONNECTION TUNE, the client should immediately send CONNECTION OPEN to prepare the connection:

```
CONNECTION OPEN
    confirm-tag             short integer
    virtual-path            short string
    client-name             short string
    options                 field table
```

confirm-tag If non-zero, the client wants a confirmation from the server. See 'Confirmation Tags' section.

virtual-path The virtual access path of the virtual host to work with. See the section on 'Virtual Servers'.

client-name The client identifier, used to identify persistent subscriptions belonging to the client. This is a string chosen by the client that uniquely defines the client. The server MUST restrict a specific client identifier to being active in at most one connection at a time.

options A set of connection options. To be defined later.

3.2.7 CONNECTION PING Command

The CONNECTION PING command implements the connection heartbeat:

```
CONNECTION PING
    respond                 bit
```

respond If 1, asks the recipient to respond with a CONNECTION PING command. Note: the response to a PING should never be a PING with respond set to 1. (For hopefully obvious reasons.)

The connection heartbeat is implemented according to the values set by each party during connection tuning: each party tell the other what kind of heartbeat it needs. The 'heartbeat' field requests a message every so often. If no message arrives within a reasonable time the client or server knows the other side has a problem, and may (for example) send a CONNECTION PING with respond set to 1, and continue or close the connection depending on the result. Or, it may just decide 'that is long enough' and close the connection.

3.2.8 CONNECTION REPLY Command

The server confirms a CONNECTION OPEN command by sending CONNECTION REPLY command to the client:

```
CONNECTION REPLY
    confirm-tag             short integer
    reply-code              short integer
    reply-text              short string
```

confirm-tag Identifies which command(s) are being confirmed.

reply-code The reply code, usually 200. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

3.2.9 CONNECTION CLOSE Command

Either party can close the connection at any time by sending the CONNECTION CLOSE command:

```
CONNECTION CLOSE
    reply-code              short integer
    reply-text              short string
```

reply-code The reply code. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

The CONNECTION CLOSE command is completed with a handshake process that works as follows:

- Either party can send CONNECTION CLOSE when they need to close the connection.
- The correct response to an unexpected CONNECTION CLOSE is to rollback any open transactions, close all destinations, close all channels, and then reply with a CONNECTION CLOSE command.
- The initiating party should then read commands until it receives a CONNECTION CLOSE in reply. It should discard any commands it receives (except CONNECTION CLOSE) and respond to any command except CONNECTION CLOSE with another CONNECTION CLOSE. This is to ensure that faulty implementations of clients or servers (which may have lost the first close command) are correctly closed.
- When it receives CONNECTION CLOSE the initiating party should rollback any open transactions, close all destinations, close all channels, and then close the network connection.

3.2.10 Command Flow Summary

The client must expect and respond to the connection commands as follows:

CONNECTION CHALLENGE Expect this after sending the 2-byte connection initiation sequence. Expect also after sending a CONNECTION RESPONSE. Respond with a CONNECTION RESPONSE. If challenge is not understood, respond with CONNECTION CLOSE.

CONNECTION RESPONSE Never expected. Respond by closing the connection (fatal error).

CONNECTION TUNE Expect after sending CONNECTION RESPONSE. Respond with CONNECTION TUNE. If tuning options are not understood, respond with CONNECTION CLOSE.

CONNECTION PING Expect at any time. Respond by sending CONNECTION PING.

CONNECTION OPEN Never expected. Respond by closing the connection (fatal error).

CONNECTION REPLY Expect after sending CONNECTION OPEN.

CONNECTION CLOSE Expect at any time. Respond by releasing all resources and sending CONNECTION CLOSE.

Physical connection close Expect at any time. Respond by releasing all (remaining) resources and closing the connection.

The server must expect and respond to the connection commands as follows:

CONNECTION CHALLENGE Never expected. Respond by closing the connection (fatal error).

CONNECTION RESPONSE Expect after sending CONNECTION CHALLENGE. Respond with CONNECTION TUNE (if response was sufficient) or CONNECTION CHALLENGE (if more information is needed). If response is not understood, respond with CONNECTION CLOSE.

CONNECTION TUNE Expect after sending CONNECTION TUNE. If tuning options are valid, no response, else respond with CONNECTION CLOSE.

CONNECTION PING Expect at any time. Respond by sending CONNECTION PING.

CONNECTION OPEN Expect after a CONNECTION TUNE. Respond with a CONNECTION REPLY if a confirmation is wanted.

CONNECTION REPLY Never expected. Respond by closing the connection (fatal error).

CONNECTION CLOSE Expect at any time. Respond by releasing all resources and sending CONNECTION CLOSE.

Physical connection close Expect at any time. Respond by releasing all (remaining) resources and closing the connection.

3.2.11 Confirmation Tags

Various commands may be optionally confirmed; that is, the client tells the server whether it wants confirmation or not. These commands all contain a field 'confirm-tag'. If the confirm-tag is zero, the server will not confirm the command. If the confirm-tag is greater than zero, the server will confirm the command. It is often easy to work without confirmations, since an error will be sharply visible as the server closes the channel or connection accordingly. However, some clients can find the confirmations useful to drive their state machines.

Confirmations are never batched: if the client requests a confirmation then the server will not send any other command back to the client until the command being confirmed has completed.

3.2.12 Connection Error Handling

It should be assumed that any exceptional conditions at this stage are due to an hostile attempt to gain access to the server. The general response to any exceptional condition in the connection negotiation is to pause that connection (presumably a thread) for a period of several seconds and then to close the network connection. This includes syntax errors, over-sized data, or failed attempts to authenticate. The server implementation should log all such exceptions and flag or block multiple failures originating with the same IP address.

3.2.13 Example

```
S: CONNECTION CHALLENGE  v1 'PLAIN TLS'
C: CONNECTION RESPONSE   'PLAIN' 'login:guest password:guest@home'
S: CONNECTION TUNE       frame-max=48000 channel-max=1024
C: CONNECTION TUNE       frame-max=16384 channel-max=1
...
S: CONNECTION CLOSE      540 'Not implemented'
C: CONNECTION CLOSE      200 'Ok'
```

3.3 Channel Commands

3.3.1 Channel Command Summary

CHANNEL OPEN Client asks server to open a new channel. Server may respond with CHANNEL REPLY, or with nothing if client did not request confirmation.

CHANNEL COMMIT Client asks server to commit the current transaction.

CHANNEL ROLLBACK Client asks server to rollback the current transaction.

CHANNEL ACK Client acknowledges one or more destination message that is has processed.

CHANNEL REPLY Server or client confirms a channel command.

CHANNEL CLOSE Either party closes the channel, the other confirms.

3.3.2 CHANNEL OPEN Command

The client opens a channel by sending CHANNEL OPEN to the server:

```
CHANNEL OPEN
    channel-id              short integer
    confirm-tag             short integer
    transacted              bit
    restartable             bit
    options                 field table
    out-of-band             short string
```

channel-id The channel ID chosen by the client, must be greater than zero.

confirm-tag If non-zero, the client wants a confirmation from the server. See 'Confirmation Tags' section.

transacted If 1, the channel operates in transacted mode. Transacted channels support the COMMIT and ROLLBACK commands. See explanation below.

restartable If 1, the channel operates in restartable mode. See explanation below.

options Channel options as name/value pairs. To be defined later.

out-of-band In case of out-of-band message transfers (using shared memory or other non-network stream methods), provides specifications of how and where the out-of-band transfers occur. The interpretation of the out-of-band string is not defined here; it is by agreement between implementations of the client and server.

CHANNEL OPEN is not allowed when the channel is already open. The server responds to a successful CHANNEL OPEN with a CHANNEL REPLY if the confirm-tag was non-zero.

3.3.2.1 Transacted Channels

AMQP supports transacted and non-transacted channels. In general transacted channels are more reliable but slower than non-transacted channels. A client can choose the transacted mode for each channel as it needs.

Transactions are done using the ANSI model; namely the transaction starts as soon as the transacted channel is opened, or a previous transaction is closed. There is no explicit 'begin' command, and there is no concept of 'nested transactions'.

3.3.2.2 Restartable Mode

Restartable mode is designed for file transfer scenarios over slow or imperfect networks. It implements a "resume" function similar to that provided by HTTP and FTP. When a large file has been partly transmitted in a past session, the original sender can resume sending from the last data sent.

To enable restartable mode, the client must set the 'restartable' option when opening a channel. In restartable mode the client MAY use the HANDLE PREPARE command to initiate an upload to the server, and the client MUST be able to handle HANDLE PREPARE commands sent by the server.

The minimum implementation of restartable mode is to discard all partial messages if transmission fails, and to answer HANDLE PREPARE commands with "this is a new message".

A full implementation of restartable mode is to hold all partial messages and to answer HANDLE PREPARE commands with "N octets of this message have been received previously", and to check the content-hash of the message after full reception.

All AMQP servers MUST support at least the minimum implementation of restartable mode. AMQP clients MAY support restartable mode or MAY choose not to use it. An AMQP server will never send HANDLE PREPARE unless the client specifically opens a channel in restartable mode. In the fully-implemented scenario, both client and server handle full restartability and content-hash validation.

Clients and servers may use any algorithm to decide when to attempt to restart a file transfer. Clients and servers MAY NOT use the HANDLE PREPARE command for messages that fit into a single fragment. The recommended algorithm is to use the HANDLE PREPARE command for messages of two or more fragments, when the channel was opened in restartable more.

Restarting a file transfer is equivalent to sending the first fragment and asking, ”does this message look familiar?” Instead of a HANDLE SEND or HANDLE NOTIFY we use HANDLE PREPARE, which includes the first fragment of data. The recipient checks whether it has a matching file, and returns a HANDLE READY command stating how much data it already has. The sender then sends the rest of the data using HANDLE SEND or HANDLE NOTIFY commands as usual.

After a message is fully transferred the recipient MAY validate the correctness of the message by recalculating the actual message’s content hash (using the SHA1 algorithm) and comparing that with the content hash supplied by the sender.

The recipient MUST reply with an error code 542 (invalid message contents) if the message hash is incorrect.

Note that restartable mode is the only area where the protocol provides functionality that is aimed at file transfer. For all other intents and purposes, files are just messages, whatever their size.

3.3.3 CHANNEL ACK Command

The client does manual acknowledgement of messages received through HANDLE NOTIFY commands (see later) using the CHANNEL ACK command:

```
CHANNEL ACK
    channel-id              short integer
    confirm-tag             short integer
    message-nbr             long integer
```

channel-id The channel being used. The channel must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

message-nbr Acknowledge up to and including this message. Note that message numbers are assigned by the server and are sequential on a single channel.

The CHANNEL ACK command acknowledges all messages received, on all open destinations, for a specific channel, up to and including the message number specified. The message specified does not need to exist.

3.3.4 CHANNEL COMMIT Command

The client sends CHANNEL COMMIT to commit work done in the current transaction:

```
CHANNEL COMMIT
    channel-id              short integer
    confirm-tag             short integer
    options                 field table
```

channel-id The channel that will be committed. The channel must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

options Transaction options as name/value pairs. To be defined later.

This command is valid only on transacted channels. When the command is complete, a new transaction starts immediately.

A commit acts on any HANDLE SEND, HANDLE NOTIFY, and CHANNEL ACK commands done on the channel since the last commit or rollback.

When the transaction is committed, any HANDLE SENDs are committed to the destinations they were sent to (which can be any mixture of queues, and topics in a single channel), and other clients consuming from those queues or topics may start to receive HANDLE NOTIFY commands. The transaction also confirms all messages sent to the client by HANDLE NOTIFY commands.

Support for XA, TIP (IETF RFC 2372) and other transaction types is provided through the use of the channel options (specified at channel open time), and will be defined later (in a further revision of this document or a separate AMQ RFC).

3.3.5 CHANNEL ROLLBACK Command

The client sends CHANNEL ROLLBACK to roll back work done in the current transaction:

```
CHANNEL ROLLBACK
    channel-id              short integer
    confirm-tag             short integer
    options                 field table
```

channel-id The channel that will be rolled back. The channel must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

options Transaction options as name/value pairs. To be defined later.

This command is valid only on transacted channels. When the command is complete, a new transaction starts immediately.

A rollback acts to cancel any HANDLE SEND, HANDLE NOTIFY, and CHANNEL ACK commands done on the channel since the last commit or rollback.

3.3.6 CHANNEL REPLY Command

The server can optionally confirm channel commands using CHANNEL REPLY:

```
CHANNEL REPLY
    channel-id              short integer
    confirm-tag             short integer
    reply-code              short integer
    reply-text              short string
```

channel-id The channel on which the original command came. The channel must be open.

confirm-tag Identifies which command(s) are being confirmed.

reply-code The reply code, usually 200. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

3.3.7 CHANNEL CLOSE Command

Either party can close a channel at any time by sending CHANNEL CLOSE:

```
CHANNEL CLOSE
    channel-id              short integer
    reply-code              short integer
    reply-text              short string
```

channel-id The channel to close. The channel must be open or in the process of closing.

reply-code The reply code. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

The CHANNEL CLOSE command is completed with a handshake process that works as follows:

- Either party can send CHANNEL CLOSE.
- The correct response to an unexpected CHANNEL CLOSE is to rollback the current transaction, if any, close all destinations opened within the channel, and then reply with a CHANNEL CLOSE command.
- The initiating party should read commands on this channel until it receives a CHANNEL CLOSE in reply. It should respond to any command within the channel, except CHANNEL CLOSE, with another CHANNEL CLOSE.
- When it receives CHANNEL CLOSE the initiating party should rollback the current transaction if any, close all destinations opened within the channel, and then consider the channel to be closed.

3.3.8 Confirmation Tags

The server confirms the CHANNEL OPEN, COMMIT, and ROLLBACK commands depending on whether the confirm-tag field is zero or not. If it is greater than zero, the server will confirm the command. If there is an error processing one of these commands the server closes the channel by sending CHANNEL CLOSE.

3.3.9 Command Flow Summary

The client must expect and respond to the channel commands as follows:

CHANNEL OPEN Never expected. Respond by closing the connection (fatal error).

CHANNEL ACK Never expected. Respond by closing the connection (fatal error).

CHANNEL COMMIT Never expected. Respond by closing the connection (fatal error).

CHANNEL ROLLBACK Never expected. Respond by closing the connection (fatal error).

CHANNEL REPLY Expect after a channel command that asks for confirmation (has a non-zero confirm-tag). No response is needed.

CHANNEL CLOSE Expect at any time on an open channel. Respond by closing the channel and reply with CHANNEL CLOSE.

The server must expect and respond to the channel commands as follows:

CHANNEL OPEN Expect on a closed channel. Reply with CHANNEL REPLY if confirmation was requested. On an open channel, reply with CHANNEL CLOSE.

CHANNEL ACK Expect on an open destination. Reply with HANDLE REPLY if confirmation was requested.

CHANNEL COMMIT Exepect on an open channel. Reply with CHANNEL REPLY if confirmation was requested and the transaction can be committed.

CHANNEL ROLLBACK Exepect on an open channel. Reply with CHANNEL REPLY if confirmation was requested and the transaction can be rolled back.

CHANNEL REPLY Never expected. Respond by closing the connection (fatal error).

CHANNEL CLOSE Expect at any time on an open channel. Respond by rolling back any open transaction, closing the channel, and reply with CHANNEL CLOSE.

3.3.10 Channel Error Handling

Errors in channel commands are most likely due to programming errors in the client: using invalid values for fields or trying to use channels in an incorrect manner. The general server response to any such error is to close the connection with a CONNECTION CLOSE command with a suitable error reply code.

3.4 Exchanging Messages

3.4.1 Message Design

By ”message” we mean the payload of the AMQP dialogue, the blocks of application data we want to carry from point to point.

The design of the message framing is intended to support both very large and ordinarily small messages, and to make it easy to read and write messages without copying the data into buffers or otherwise scanning and processing it.

3.4.2 Message Fragmentation

Messages that are longer than a certain size (this size is actually the maximum frame size minus the size of a message's header) are split into fragments and each fragment is sent with a separate command. On a given channel the fragments for a message are sent as a serial stream. It is not allowed to mix fragments from different messages on the same channel. At the connection level, however, fragments from different messages can be interleaved, each message being on a distinct channel.

3.4.3 Message Frames

AMQP frames messages outside commands. Looking at the TCP/IP stream we might see something like this (using somewhat abstract command names):

```
[OPEN][SEND][message][SEND][message][CLOSE]
```

Messages come after specific commands, namely HANDLE SEND and HANDLE NOTIFY. These commands carry a message from client to server and from server to client respectively.

In the HANDLE SEND and HANDLE NOTIFY commands we see four fields that control exactly how the message is framed:

```
    ...
    fragment-size           long integer
    partial                 bit
    out-of-band             bit
    recovery                bit
    ...
```

The fragment-size field is the size of the message frame that follows. It's called 'fragment-size' because the message frame can contain part of a message, a fragment. The partial field indicates whether the message is split into fragments; it will be zero on messages that are not fragmented, or on the last fragment of a message.

The out-of-band field indicates that the client and server are exchanging message data in some other way. So if this option is used, the TCP/IP stream would look like this:

```
[OPEN][SEND][SEND][CLOSE]
```

Finally, the recovery field is used when transferring very large messages across unreliable lines; it just means 'I'm sending that part of the message you don't already have'. Recovery relies on the HANDLE PREPARE command.

3.4.4 Message Header

The message has two sections, a header block and a body block:

```
[   header block   ][   body block   ]
|<-------- size of fragment -------->|
```

The header block holds this data (all integer data is in network byte order):

```
MESSAGE
    body-size               long integer
    persistent              bit
    priority                octet
    expiration              long integer
    mime type               short string
    encoding                short string
    identifier              short string
    headers                 field table
```

body-size The length of the content block.

persistent If 1, the message is held on safe storage and will be delivered even if the server suffers a power failure or other crash. If 0, the message may be held only in memory. This flag lets the client choose between reliability and performance.

priority A value from 0 to 9, where 0-4 is normal priority and 5-9 is high priority. The server will do its best to deliver high priority messages ahead of normal ones.

expiration A time value in POSIX time_t format, after which the message can be expired by the server. The resolution of this field is 1 second, which is the assumed accuracy of clock synchronization on an AMQ network. Note that the time_t format is always UTC. This field can be zero, in which case the message will not be expired.

mime-type The MIME content type of the message. If empty, defaults to the mime-type value specified in the open command for the type of destination being used.

encoding The content encoding of the message. If empty, defaults to the msg-encoding value specified in the open command for the type of destination being used.

identifier The application-defined identifier for the message.

headers A set of application-defined headers which can be used for message matching and processing.

The body block holds zero or more octets of message data, formatted in accordance with the MIME type and encoding. The maximum size of a message as carried by AMQP/Fast is 4Gb. The destination may implement its own fragmentation scheme using the identifier, i.e. splitting a 8Gb message into two parts.

The procedure to write a message is:

1. Build the message header block and prepare or locate the message body block.
2. Calculate the combined size of the two blocks. This is the amount of data to be written to the connection, the full message size.
3. Partition the combined block into fragments (not literally, but using calculated offsets) and write each fragment to the connection using an appropriate command.
4. The intermediate commands have 'partial' set to 1, the final one has 'partial' set to zero.

The procedure to read a message is:

1. Parse the command frame and read the following fragment.
2. If this is the first fragment, decode the header and store the length of the body block.
3. Data following the header, in the first fragment, is the first part of the message body block.
4. Read successive message-carrying commands and for each one, read the fragment data to construct the full message.

3.5 Destination Commands

3.5.1 Destination Command Summary

These are destination commands the client may send to the server:

HANDLE OPEN Client asks server to prepare to use a specified destination or set of destinations. Client can open temporary queues and topics.

HANDLE SEND Client sends a message to a specific destination.

HANDLE CONSUME Client waits for a message from a specific destination.

HANDLE FLOW Client tells server to stop/restart messages from a specific destination.

HANDLE UNGET Client pushes a message back onto the destination (for queues only).

HANDLE QUERY Client asks for a list of all messages waiting on a destination.

HANDLE BROWSE Client asks server for a specific message without marking it as being delivered to that client.

These are destination commands the server may send to the client:

HANDLE CREATED Server informs client that a temporary queue or topic is now available.

HANDLE NOTIFY Server sends a message from a destination back to the client.

HANDLE INDEX Server provides client with the list of messages waiting on the destination.

HANDLE REPLY Server confirms a destination command.

These are destination commands that either party may send:

HANDLE PREPARE Client asks server, or server asks client, to prepare space for a large message.

HANDLE READY Server tells client that a specified message can be transferred.

HANDLE CLOSE Either party closes access to the destination, the other confirms.

3.5.2 HANDLE OPEN Command

The client opens access to a destination by sending HANDLE OPEN to the server:

```
HANDLE OPEN
    channel-id              short integer
    handle-id               short integer
    service-type            short integer
    confirm-tag             short integer
    producer                bit
    consumer                bit
    browser                 bit
    temporary               bit
    dest-name               short string
    mime-type               short string
    encoding                short string
    options                 field table
```

channel-id The channel ID used for all subsequent commands using this handle. Note that the channel ID is specified only in the HANDLE OPEN command. Further commands using the handle do not specify the channel-id.

handle-id The handle reference chosen by the client. This ID must be unique within the connection and must not refer to a handle that is already open.

service-type The service type required. The valid service types are defined below.

confirm-tag If non-zero, the client wants a confirmation from the server.

producer If 1, the client requests access to post messages to the destination. Used for queues and topics only.

consumer If 1, the client requests access to consume messages from the destination (I.E. to listen to the destination).

browser If 1, the client requests access to browse messages from the destination.

temporary If 1, the client asks the server to create a temporary destination as specified. See explanation of temporary destinations below.

dest-name The name of the destination to open. May be empty, see section on 'Destination Name Hierarchies'.

mime-type The default MIME content type for HANDLE SEND commands sent to the destination. This can be overridden on a per-message basis. If empty, defaults to 'application/octet-stream'.

encoding The default content encoding for HANDLE SEND commands sent to the destination. This can be overridden on a per-message basis. If empty, defaults to 'binary'.

options Handle options as name/value pairs. To be defined later.

A handle is always part of a channel, so if the client opens channel 5, then handle 20 on channel 5, it can use handle 20 thereafter without having to say "and channel 5" each time. This is why the other handle commands don't have a channel id,

3.5.2.1 AMQP Service Types

The valid service types are:

%d01 Queue service.

%d02 Topic service.

Destinations are managed at the server end, and the specific implementation of any specific service type is hidden from the client.

3.5.2.2 Temporary Queues and Topics

Temporary queues (and temporary topics, which work exactly the same way) **PH: 'exactly the same way'? are designed to provide clients with a place for other processes to send replies. Typically, a client will create a temporary queue and then ask other processes to "reply to" that queue. The value of a temporary queue is that it is deleted automatically; when the connection closes, the queue is destroyed.

Temporary queue names may be specified by the client, or may be chosen by the server. If the client provides a queue name, the server verifies that this queue either does not exist, or has previously existed with the same client as owner. Reopening a temporary queue is equivalent to purging it; any messages it contains are deleted. If the client does not specify a queue name the server will create a unique queue name. The server returns the queue name to the client in a HANDLE CREATED command.

If the client also specified a confirm-tag, the HANDLE REPLY comes after the HANDLE CREATED command.

Only the client that created a temporary queue can consume messages from it.

3.5.2.3 Destination Name Hierarchies

AMQP explicitly supports name hierarchies for queue and topic names:

```
alt.rec.pets
alt.rec.pets.cats
alt.rec.pets.cats.discuss
alt.rec.pets.dogs
alt.rec.pets.dogs.discuss
```

The HANDLE OPEN and HANDLE SEND commands specify part or all of a destination name. Subsequent commands on the same destination provide further refinement of the name through simple concatenation of the name provided in HANDLE OPEN and the name provided in HANDLE SEND.

The HANDLE OPEN command MAY refer to a full and valid destination name, or it MAY refer to a name prefix shared by a set of destinations.

Here is how a client writes a message to all destinations in a hierarchy:

```
open (empty name)
send (empty name)
```

Here is how a client writes a message to different destinations within a hierarchy:

```
open alt.rec.
send pets
send pets.cats.discuss
```

Here is the client attempting to write to non-existent destinations, an error that would provoke a HANDLE CLOSE command from the server:

```
open alt.rec.pets
send alt
send pets
send alt.rec.pets.cats
```

Note that dots are not special; destination names can be delimited by any character that is valid in a destination name, which excludes /, #, &, and some other special characters. For details see AMQ RFC 013.

3.5.3 HANDLE SEND Command

To send a message to a destination the client sends a HANDLE SEND command:

```
HANDLE SEND
    handle-id               short integer
    confirm-tag             short integer
    fragment-size           long integer
    partial                 bit
    out-of-band             bit
    recovery                bit
    dest-name               short string
    [message fragment]
```

handle-id The handle reference chosen by the client, must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

fragment-size The amount of data provided after the frame; the size of the fragment.

partial A message is sent as one or more fragments. This field indicates whether this is the final fragment for the message. If set to zero, this is the final or only fragment. Message fragments are sent consecutively on a particular channel and with the same tag.

out-of-band Indicates whether the message data is transferred out-of-band or not. If set, the server should not expect to read the message fragment from the connection stream, but from some other location as defined by the out-of-band field sent during the HANDLE OPEN command.

recovery If 1, tells the server that the message is being sent partially, from the recovery point indicated by the server in the HANDLE READY command (used in restartable operations only).

dest-name The name of the destination to write to. May be empty, see section on 'Destination Name Hierarchies'. Note: this field is ignored when 'partial' is 1. I.E. you should set it or expect it only on the last fragment of a message.

Note that the message is sent immediately after the HANDLE SEND frame, on the same channel. The message consists of a header and a body which can be split into multiple fragments. These fragments follow one another on the same channel. Multiple messages can be interleaved, but on their own distinct channels.

In restartable mode, after a HANDLE PREPARE/HANDLE READY exchange, the HANDLE SEND command carries that part of the message not already held by the server. The message header is always carried by the HANDLE PREPARE command.

3.5.4 HANDLE CONSUME Command

The client asks to receive messages from a specific destination by sending the HANDLE CONSUME command:

```
HANDLE CONSUME
    handle-id               short integer
    confirm-tag             short integer
    prefetch                short integer
    no-local                bit
    no-ack                  bit
    dest-name               short string
    selector                field table
```

handle-id The handle reference chosen by the client, must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

prefetch The number of outstanding messages the client can handle at once. The server will send up to this many messages without receiving a CHANNEL ACK from the client. The minimum value for this field is 1, the maximum is 65535.

no-local If 1, the client will not receive messages sent on the same connection. This field is used only for topic services.

no-ack If 1, the client will not acknowledge messages received from this destination. The client can specify 'no ack' when it wants the highest possible throughput and can tolerate lost messages.

dest-name The name of the destination to listen to. May be empty, see section on 'Destination Name Hierarchies'.

selector The fields to select on, if any. Fields are selected on by presence and value - an empty field value means "present", any non-empty value means "this value". Field selectors are always inclusive, i.e. ALL fields must match for a message to be passed to the handle.

The HANDLE CONSUME command creates a subscription when used on a topic. Subscriptions last until the client closes the handle, channel, or connection.

3.5.5 HANDLE FLOW Command

The client asks the server to pause or restart messages from a destination:

```
HANDLE FLOW
    handle-id               short integer
    confirm-tag             short integer
    flow-pause              bit
```

handle-id The handle reference chosen by the client, must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

flow-pause If 1, the server stops sending messages to the client. If 0, the server restarts sending messages to the client.

By default all handles are set to ”ON”, that is, a client does not need to use HANDLE FLOW to start handles, only to pause or restart messages coming from a destination.

The HANDLE FLOW command applies to all the destinations covered by the HANDLE OPEN command.

For topics, the HANDLE FLOW command acts to pause the sending of HANDLE NOTIFY messages. For queues, the HANDLE FLOW command causes messages to be sent to other clients (if there are others consuming from the same queue).

3.5.6 HANDLE UNGET Command

For queue services only, the client can push a message back to its original destination by sending the HANDLE UNGET command:

```
HANDLE UNGET
    handle-id               short integer
    confirm-tag             short integer
    message-nbr             long integer
```

handle-id The handle reference chosen by the client, must be open.

confirm-tag If non-zero, the client wants a confirmation from the server.

message-nbr A server message ID: this must be used in any command that refers to the message.

The HANDLE UNGET command tells the server that the client was unable to process the message and that it should be passed to another consuming client. Note that this can cause messages to be delivered out of order. Thus when more than one client is consuming messages from a queue, the order of message delivery is no longer guaranteed.

This command MUST be sent before any CHANNEL ACK that covers or refers to the same message. For example, if the client receives messages numbered 1, 2, 3, 4 and wishes to unget message 3 and acknowledge the remaining messages, it should send ’unget 3’ followed by ’ack 4’. It can also send ’ack 1’, ’ack 2’, ’unget 3’, ’ack 4’.

3.5.7 HANDLE QUERY Command

The client can ask the server for a list of all outstanding messages on a destination by sending the HANDLE QUERY command:

```
HANDLE QUERY
    handle-id               short integer
    message-nbr             long integer
    dest-name               short string
    selector                long string
    mime-type               short string
```

handle-id The handle reference chosen by the client, must be open.

message-nbr The client is interested in all messages with an ID greater or equal to this value. When zero, means ”all messages” (a message can have an ID value zero).

dest-name The name of the destination to examine. May be empty, see section on 'Destination Name Hierarchies'.

selector A detailed specification of the precise messages that the client wants. The precise selector syntax is implementation-defined and the MIME type tells the server which implementation to use.

mime-type The MIME type for the selector. Standard MIME types are defined later.

3.5.8 HANDLE BROWSE Command

The client asks to look at (but not consume) a single message using HANDLE BROWSE:

```
HANDLE BROWSE
    handle-id               short integer
    confirm-tag             short integer
    message-nbr             long integer
```

handle-id The handle reference chosen by the client, must be open.

confirm-tag Should be non-zero if the client sends multiple HANDLE BROWSE commands without waiting for a server response (so that the client can match HANDLE REPLY responses for 'message not found').

message-nbr One of the server message IDs provided by the HANDLE INDEX command.

There is no guarantee that the message will still be available when the client asks for it. If the message was consumed by another client, the server replies a HANDLE REPLY command with reply code 310 (message not found). Otherwise the server replies with HANDLE NOTIFY (with the delivered flag set to zero).

3.5.9 HANDLE CREATED Command

When the server creates a temporary destination it replies to the client with a HANDLE CREATED command:

```
HANDLE CREATED
    handle-id               short integer
    dest-name               short string
```

handle-id The destination ID chosen by the client in the HANDLE OPEN command.

dest-name The full name of the temporary destination created. This name will be unique within the virtual host.

3.5.10 HANDLE NOTIFY Command

When the server has a message to send to the client it sends a HANDLE NOTIFY command:

```
HANDLE NOTIFY
    handle-id               short integer
    message-nbr             long integer
    fragment-size           long integer
    partial                 bit
    out-of-band             bit
    recovery                bit
    delivered               bit
    redelivered             bit
    dest-name               short string
    [message fragment]
```

handle-id The handle reference sent by the client in the HANDLE CONSUME or HANDLE BROWSE command.

message-nbr A server message ID: this must be used in any command that refers to the message.

fragment-size The amount of data provided after the frame; the size of the fragment.

partial Indicates whether this is the final fragment for the message, 1 = more fragments, or 0 = final or only fragment. Message fragments are sent consecutively on the current channel.

out-of-band Indicates whether the message data is transferred out-of-band or not. If set, the client should not expect to read the message fragment from the connection stream, but from some other location as defined by the out-of-band field sent during the HANDLE OPEN command.

recovery If 1, tells the client that the message is being sent partially, from the recovery point indicated by the client in the HANDLE READY command (used in restartable operations only).

delivered If 1, the message is being delivered to the client and the client should process it accordingly and acknowledge the message with CHANNEL ACK when it can. This field will be zero when the client asked for the message using HANDLE BROWSE, and 1 when the client used HANDLE CONSUME. The client MUST NOT process messages with the delivered field set to zero.

redelivered If 1, the server considers the message to have already been delivered to the client at some past time. This is a hint which the client can use to double-check that it has not already processed the message.

dest-name The full name of the destination from which the message came. Note: this field is not set when 'partial' is 1. I.E. you expect it only on the last fragment of a message.

Note that the message is sent immediately after the HANDLE NOTIFY frame, on the same channel. The message consists of a header and a body which can be split into multiple fragments. These fragments follow one another on the same channel. Multiple messages can be interleaved, but on their own distinct channels.

In restartable mode, after a HANDLE PREPARE/HANDLE READY exchange, the HANDLE NOTIFY command carries that part of the message not already held by the client. The message header is always carried by the HANDLE PREPARE command.

3.5.11 HANDLE INDEX Command

The server responds to a HANDLE QUERY command with a HANDLE INDEX command:

```
HANDLE INDEX
    handle-id               short integer
    message-nbr             long integer
    message-list            long string
```

handle-id The handle reference sent by the client in the HANDLE QUEUE command.

message-nbr If non-zero, this is the highest message-nbr contained in the message list and there are further messages available to inspect.

message-list A list of message ID's available. This list is formatted as an ASCII string delimited by spaces and hyphens, E.G. "0 5 9 15-40 80".

The lowest valid message ID is zero. The highest valid message ID is 2^32-1. Note that message IDs are not permanent - the server can assign these on any basis so long as it can resolve the message ID used in a HANDLE BROWSE command.

There is an implicit limit to how many message numbers the server will return; this limit depends on the server implementation. If there are more messages available than can be returned in the command the server sets the message-nbr field to the highest number returned in the list.

3.5.12 HANDLE PREPARE Command

The HANDLE PREPARE command is used for restartable transfers. Either the server or the client can send it in advance of sending a large message. The command has two functions: it verifies that enough space is available to receive the message, and it allows for recovery of partially-sent messages (similar to a restarted download or upload).

The HANDLE PREPARE command carries the message header plus zero or more octets of body data, as for a HANDLE SEND command:

```
HANDLE PREPARE
    handle-id               short integer
    fragment-size           long integer
    content-hash            short string
    [message header]
```

handle-id The handle reference chosen by the client, must be open.

fragment-size The amount of data provided after the frame; the size of the message header plus zero or more octets of body data.

content-hash The SHA1 hash of the message contents. The SHA1 algorithm provides a standard cryptographic hash. The recipient (server or client) will store this hash and when the message has been fully received, recalculate and check it. May be empty, in which case no validation will be done.

The identity of a message is uniquely defined by the complete contents of its header. That is, two messages headers are considered to refer to the same message only when every field in the header is identical - including such fields as expiration, priority, and persistence.

The client and server MAY NOT send multiple HANDLE PREPARE commands in batches since there is no mechanism to distinguish returning HANDLE READY commands.

3.5.13 HANDLE READY Command

The party receiving HANDLE PREPARE answers with HANDLE READY:

```
HANDLE READY
    handle-id              short integer
    message-size           long integer
```

handle-id The handle reference chosen by the client, must be open.

message-size The amount of message data already stored, in case the message was previously transferred and the transfer broken off. In the case of new messages, will be the size of the body data provided in the HANDLE PREPARE command.

Note that messages are identified uniquely by their header. The server or client receiving large messages is expected to hold these in a temporary area until completely received. The HANDLE PREPARE command inspects this temporary area to find a matching message; if it finds one, it reports the amount of data already received.

A client that requests restartable mode from the server must be able to support the HANDLE PREPARE command. Naive servers and clients may respond with a HANDLE READY with message size of zero.

If the recipient cannot accept the message as specified, it closes the channel with the reply code 311 (message too large). There is no recovery for oversized messages.

3.5.14 HANDLE REPLY Command

The server may confirm destination commands - when the client asks for this - using HANDLE REPLY:

```
HANDLE REPLY
    handle-id              short integer
    confirm-tag            short integer
    reply-code             short integer
    reply-text             short string
```

handle-id The destination ID chosen by the client, to which the command was sent.

confirm-tag Identifies which command(s) are being confirmed.

reply-code The reply code. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

3.5.15 HANDLE CLOSE Command

Either party can close a handle at any time by sending HANDLE CLOSE:

```
HANDLE CLOSE
    handle-id               short integer
    reply-code              short integer
    reply-text              short string
```

handle-id The destination ID chosen by the client, referring to an open destination.

reply-code The reply code. The AMQ reply codes are defined in AMQ RFC 011.

reply-text The localised reply text.

The HANDLE CLOSE command is completed with a handshake process that works as follows:

- Either party can send HANDLE CLOSE.
- The initiating party should read and process commands on this channel until it receives a HANDLE CLOSE in reply. If it sent HANDLE CLOSE due to an error, it can discard the commands it receives.
- When it receives HANDLE CLOSE the initiating party can consider the channel closed, and release resources.
- The correct response to an unexpected HANDLE CLOSE is to reply with HANDLE CLOSE and then to treat the handle as closed.

3.5.16 Command Flow Summary

The client must expect and respond to the destination commands as follows:

HANDLE OPEN Never expected. Respond by closing the connection (fatal error).

HANDLE SEND Never expected. Respond by closing the connection (fatal error).

HANDLE CONSUME Never expected. Respond by closing the connection (fatal error).

HANDLE FLOW Never expected. Respond by closing the connection (fatal error).

HANDLE UNGET Never expected. Respond by closing the connection (fatal error).

HANDLE QUERY Never expected. Respond by closing the connection (fatal error).

HANDLE BROWSE Never expected. Respond by closing the connection (fatal error).

HANDLE CREATED Expect after opening a temporary destination. Store the destination name.

HANDLE NOTIFY Expect on an open destination after sending HANDLE CONSUME or HANDLE BROWSE. Store the message (or message fragment). If the message is being 'delivered', reply with CHANNEL ACK either immediately, or after the message has been processed by the application.

HANDLE INDEX Expect on an open destination after sending HANDLE QUERY.

HANDLE REPLY Expect after a destination command that asks for confirmation (has a non-zero confirm-tag). No response is needed.

HANDLE PREPARE Expect on an open destination. Reply with HANDLE READY command.

HANDLE READY Expect after sending HANDLE PREPARE. Reply with HANDLE SEND.

HANDLE CLOSE Expect at any time on an open destination. Respond by replying with HANDLE CLOSE and closing the destination.

The server must expect and respond to the destination commands as follows:

HANDLE OPEN Expect on a closed destination. Reply with HANDLE REPLY if confirmation was requested. On an open destination, reply with HANDLE CLOSE.

HANDLE SEND Expect on an open destination. Reply with HANDLE REPLY if confirmation was requested.

HANDLE CONSUME Expect on an open destination. Reply with HANDLE REPLY if confirmation was requested.

HANDLE FLOW Expect on an open destination. Reply with HANDLE REPLY if confirmation was requested.

HANDLE UNGET Expect on an open destination. Reply with HANDLE REPLY if confirmation was requested.

HANDLE QUERY Expect on an open destination. Reply with HANDLE INDEX command.

HANDLE BROWSE Expect on an open destination. Reply with HANDLE SEND command.

HANDLE CREATED Never expected. Respond by closing the connection (fatal error).

HANDLE NOTIFY Never expected. Respond by closing the connection (fatal error).

HANDLE INDEX Never expected. Respond by closing the connection (fatal error).

HANDLE REPLY Never expected. Respond by closing the connection (fatal error).

HANDLE PREPARE Expect on an open destination. Reply with HANDLE READY command.

HANDLE READY Expect after sending HANDLE PREPARE. Reply with HANDLE SEND.

HANDLE CLOSE Expect at any time on an open destination. Respond by replying with HANDLE CLOSE and closing the destination.

3.6 Alternatives

Possible future alternatives to this protocol include AMQP/HTTP, a proposed implementation over HTTP, and AMQP/BEEP, a proposed implementation over BEEP. SMQP (Simple Message Queueing Protocol) is a proposed IETF standard that also covers some of the same ground, mainly with respect to topics.

3.7 Security Considerations

The AMQP/Fast framing mechanism does not provide any protection against attack per-se. However, the security profiles used provide varying degrees of assurance for authentication and privacy. To protect against a man-in-the-middle attack in which attacker replaces a high-security profile with a lower one, a server or client may be configured to refuse to open a channel without an acceptable level of security.

Application-level access is the responsibilty of the server and client.

We guard against buffer-overflow exploits by using length-specified buffers in all places. All externally-provided data can be verified against maximum allowed lengths whenever any data is read.

Invalid data can be handled unambiguously, by closing the channel or the connection.

3.8 Change Management Considerations

1. We expect to make minor refinements to this protocol before it is released into the wild.
2. We allow for a small number of version upgrades through the use of the protocol-version octet.
3. We allow clients and servers to negotiate the protocol versions that they support.
4. We allow the addition of a small number of commands without renumbering of existing commands by leaving unused slots.
5. We provide implementation-defined options in many commands.

3.9 References

AMQ RFC 008 defines general terminology. AMQ RFC 011 defines the AMQP reply codes. AMQ RFC 013 defines the AMQ URL syntax. IETF RFC 2222 defines the SASL protocol.

4 Comments on this Document

4.1 Date, name

No comments at present.