AMQ RFC 006
The AMQP/Fast Wire-Level Protocol

version 0.9a2

iMatix Corporation <amq@imatix.com>

Revised: 2005/06/06

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

2005/06/06 - 0.9a2:

- Full definition of Proxy and Queue classes and updating of all other classes and methods to remove Destination and Subscription concepts.

2005/06/05 - 0.9a1:

- Formulation of "proxy" concept for matching and routing; removed topics and subscriptions from formal protocol specificatons.

2005/05/28 - 0.9a0:

- Refactored the protocol using new framing model and protocol class/method architecture. First provisional draft specification.

1.2 Copyright Notice

1.3 Authors

AMQP was designed by Pieter Hintjens with help from John O'Hara, Mark Atwell, and others. This document was written by Pieter Hintjens. The diagrams were drawn by Gustavo Broos.

1.4 Abstract

We propose a wire-level protocol and a set of formal semantics for building large scale and high-performance middleware architectures. AMQP/0.9 is the third major design iteration of the AMQ protocol.

2 Introduction

2.1 Purpose and Goals of AMQP/Fast

AMQP/Fast is designed to be the high-performance member of the AMQP family of standardised general-purpose message-oriented middleware wire-level protocols.

A "wire-level protocol" can be a simple as a "this is how we define packets on the wire". AMQP/Fast is more ambitious than that. It tries to provide formal answers to a series of questions that we must answer in order to implement useful messaging between protocols.

We want to be able to:

- Construct logical "messages" out of data of any type and size, and move these message efficiently across a network of clients and servers.
- Provide queueing and routing semantics for messages so that the behaviour of the server is unambiguously defined by the protocol.
- Define explicit operational semantics for common messaging models, including JMS, file transfer, and streaming, at the protocol level.
- Define quality-of-service levels semantics so client applications can explicitly choose appropriate trade-offs between speed and reliability.
- Create a fully-asynchronous network so that messages can be queued when clients are disconnected, and forwarded when they are connected.
- Allow interoperation and bridging with other middleware systems.
- Allow implementation in any language, on any kind of hardware.
- Avoid license and patent concerns that might hinder adoption of the protocol as a standard.
- Allow the marshalling layers of protocol implementations to be easily generated using code generation technology.
- Allow the creation of an abstract network where services, and their data, can move around the network opaquely to the clients.

2.2 Intended Operating Lifespan

AMQP/Fast is designed to give a useful lifespan of 50 years or more. Our goal is that an AMQP/Fast peer will be able to operate continuously with no upgrades or incompatibility for at least this duration, without requiring "legacy support". One should be able to build a client or server into physical infrastructure.

While the protocol version may and will change, the protocol mechanics (framing, method structures, etc.) must operate unchanged for the full intended lifespan of the protocol, allowing full and perfect forwards compatability with all future versions of the protocol.

We have applied "Moore's Law" - the theory of exponential growth of capacity of technology - to all capacity limits to identify and eliminate potential future bottlenecks, specifically for:

1. Message sizes: the largest messages (files) are today around 20GB. We expect this to grow by 50% per year, reaching the limit defined by a 64-bit size in 50 years.

2. Frame sizes: IPv4 is limited to 64KB frames, IPv6 to 4GB frames. Ethernet itself is limited to 12000 byte frames due to its CRC algorithm. We expect the maximum networking frame size to grow by 50% per year, in large leaps. We will thus reach the limit of 64-bit sized packets in 50 years.
3. Sequencing: we use method sequencing to allow matching of replies and errors with requests, so called "synchpoints". The method sequence number is 32 bits. This allows 4G methods to be exchanged between synchpoints. We do not see this as a limitation in any scenario, present or future.
4. Protocol classes and methods: the current protocol defines about ten classes and about ten or fewer methods per class. We expect that new versions of the protocol will add classes and methods at a constant rate of 1 to 5 per year. The limit of 255 classes and methods per class should be sufficient to last until 2055.
5. Channels: the limit of 4G channels allows growth of 50% per year from an estimated usage of 10 channels per connection today.
6. Timestamps: we use 64-bit time-stamp values.

2.3 An Extensible Functionality

AMQP/Fast is intended to be extensible in several directions, including new directions totally outside the scope of the protocol as it is designed today. These are the aspects of AMQP/Fast that have been deliberately designed to be extensible (in order of increasing generality and power):

1. Adding new properties to content domains.
2. Adding new arguments to methods.
3. Adding new methods to classes.
4. Adding new content domains.
5. Adding new classes.
6. Adding new frame types.
7. Adding new protocols.

All of these should be feasible while maintaining full backwards compatability with existing implementations.

2.4 Ease of Implementation

Our goal is to achieve some of the cost-benefit ratio of protocols such as SMTP and HTTP, where a simple client can be trivial to build but a full client can be very sophisticated. Keeping AMQP/Fast accessible to simple clients is possible if we hold to these design rules:

1. The use of all complex functionality (e.g. more sophisticated data types or structures) must be optional.
2. The protocol must be able to operate entirely synchronously, since an asynchronous model - though efficient and reliable - is a barrier for simple implementations.
3. The protocol must be formally defined so that significant parts of a client or server protocol interface can be mechanically generated using code generation techniques.

2.5 Guidelines for Implementors

- We use the terms MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY as defined by IETF RFC 2119. Where possible we note the security implications of the guidelines.

- We use the term "the server" when discussing the specific behaviour required of a conforming AMQP/Fast server.
- We use the term "the client" when discussing the specific behaviour required of a conforming AMQP/Fast client.
- We use the term "the peer" to mean "the server or the client".
- All numeric values are decimal unless otherwise indicated.
- Protocol constants are shown as uppercase names. AMQP/Fast implementations SHOULD use these names when defining and using constants in source code and documentation.
- Property names, method arguments, and frame fields are shown as lowercase names. AMQP/Fast implementations SHOULD use these names in source code and documentation.
- Protocol methods are defined using a formal grammar (ASL) as defined in AMQ RFC xxx.

2.6 Glossary of Terms

These terms have significance within the context of AMQP:

Connection A network connection, e.g. a TCP/IP socket connection.

Channel A bi-directional stream of communications between two AMQP/Fast peers. Channels are multiplexed so that a single network connection can carry multiple channels.

Client The initiator of an AMQP/Fast connection or channel. AMQP/Fast is not symmetrical. Clients produce and consume messages while servers queues and route messages.

Server The party that accepts client connections and implements the AMQ queueing and routing functions.

Peer Either party in an AMQP/Fast connection. An AMQP/Fast connection involves exactly two peers (one is the client, one is the server).

Frame A formally-defined package of connection data. Frames are always written and read contiguously - as a single unit - on the connection.

Extended_Frame A frame capable of using more channels and carrying more data than a normal frame.

Meta_Frame A specific type of frame that has the sole purpose of indicating that the next frame will be an extended frame.

Protocol_Class A collection of methods that deal with a specific type of functionality.

Method A specific type of frame that passes instructions from one peer to the other.

Content Application data passed from client to server and from server to client. AMQP/Fast content can be structured into multiple parts.

Header A specific type of frame that describes content.

Body A specific type of frame that contains raw application data. Body frames are entirely opaque - the server does not examine or modify these in any way.

Trace_Frame A specific type of frame that carries information to a "trace handler", an abstract processing unit that may be embedded in an peer.

Message A specific type of content, the application-to-application message.

Proxy The entity within the server which receives messages from producer applications and decides how to forward these to queues within the server.

Queue A named entity that holds messages and forwards them to consumer applications. Queues can take many forms: transient, persistent, private, shared, etc.

Durable A server resource that survives a server restart.

Transient A server resource that is wiped or reset after a server restart.

Persistent A message that the server holds on reliable disk storage and should not lose after a server restart.

Non-persistent A message that the server holds in memory and may lose after a server restart.

Consumer A client application that requests messages from a queue.

Producer A client application that publishes messages to a proxy.

Virtual_host A collection of proxies, queues, consumers, and associated objects. Virtual hosts are independent server domains that share a common authentication and encryption environment. I.e. the client application choses the virtual host after logging in to the server.

Realm A set of server resources (proxies and queues) covered by a single security policy and access control. Applications ask for access rights for specific realms, rather than for specific resources.

Streaming The process by which the server will send messages to the client at a pre-arranged rate.

Staging The process by which a peer will transfer a large message to a temporary holding area before formally handing it over to the recipient. This is how AMQP/Fast implements restartable file transfers.

Synchpoint A synchronous confirmed step in what is otherwise an asynchronous exchange of methods.

Out-of-band_transport The technique by which data is exchanged outside the network connection. For example, two peers can exchange frames across a TCP/IP connection and then switch to using shared memory if they discover they are on the same system.

Zero_copy The technique of transferring data without copying it to or from intermediate buffers. Zero copy requires that the protocol allow the transfer of data as opaque blocks, which AMQP/Fast does.

Assertion A condition that must be true for processing to continue.

Exception A failed assertion, handled by closing either the channel or the connection.

These terms have no special significance within the context of AMQP:

Topic Usually a means of distributing messages; AMQP implements topics using one or more types of proxy.

Subscription Usually a request to receive data from topics; AMQP implements subscriptions as queues.

2.7 Limitations

These limitations are formally part of the AMQP/Fast specifications:

- Number of channels per connection, normal frames: 255.
- Number of channels per connection, extended frames: 64K-1.
- Size of a short string: 0 to 255 octets.
- Size of a long string: 0 to 64K-1 octets.
- Size of a normal frame: 0 to 64K-1 octets.
- Size of an extended frame: 0 to 2e64-1 octets.
- Number of possible content domains: 255.
- Number of protocol classes: 255 per peer.
- Number of methods: 255 per protocol class.
- Number of methods sent between synchpoints: 4G.

3 Design Proposal

3.1 Data Type Representation

3.1.1 Goals and Principles

We use a small set of basic data types that are guaranteed to work on all platforms and be easily implemented in all languages. More sophisticated data types can be packaged using the basic AMQP/Fast data types.

3.1.2 Formal Grammar for AMQP/Fast Fields

This formal grammar defines the AMQP/Fast data types:

```
amqp-field          = BIT
                    / OCTET
                    / short-integer
                    / long-integer
                    / long-long-integer
                    / short-string
                    / long-string
                    / time-stamp
                    / field-table
short-integer       = 2*OCTET
long-integer        = 4*OCTET
long-long-integer   = 8*OCTET
short-string        = OCTET *string-char
string-char         = %d1 .. %d255
long-string         = short-integer *OCTET
time-stamp          = long-long-integer
field-table         = long-integer *field-value-pair
field-value-pair    = field-name field-value
field-name          = short-string
field-value         = 'S' short-string
                    / 'L' long-string
                    / 'I' signed-integer
                    / 'D' decimal-value
                    / 'T' time-stamp
                    / 'F' field-table
signed-integer      = 4*OCTET
decimal-value       = decimals long-integer
decimals            = OCTET
```

Guidelines for implementors:

- A peer MUST support all the above data types.

3.1.3 Integers

AMQP/Fast defines these integer types:

- Bit values, which are accumulated into octets.
- Unsigned octet (8 bits).
- Unsigned short integers (16 bits).
- Unsigned long integers (32 bits).
- Unsigned long long integers (64 bits).

Integers and string lengths are always unsigned and held in network byte order. We make no attempt to optimise the case when two low-high systems (e.g. two Intel CPUs) talk to each other.

Guidelines for implementors:

- Implementors MUST NOT assume that integers encoded in a frame are aligned on memory word boundaries.

3.1.4 Strings

All strings are variable-length and represented by an integer length followed by zero or more octets of data. AMQP/Fast defines these string types:

- Short strings, stored as a 1-octet length followed by zero or more octets of data. Short strings are capable of carrying UTF-8 data, and may not contain binary zero octets. In the current version of the protocol short strings may only contain US-ASCII (ISO-8859-1) characters.
- Long strings, stored as a short integer length followed by zero or more octets of data. Long strings can contain any data.

3.1.5 Timestamps

Time stamps are held in the 64-bit POSIX time_t format with an accuracy of one second.

3.1.6 Field Tables

Field tables 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. A field can hold a tiny text string, a short binary string, a long signed integer, a decimal, a date and/or time, or another field table.

Guidelines for implementors:

- Field names MUST start with a letter, '$' or '#' and may continue with letters, '$' or '#', digits, or underlines, to a maximum length of 128 characters.
- A peer SHOULD validate field names.
- Specifically and only in field tables, integer values are signed (31 bits plus sign bit).
- 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.
- A peer MUST handle duplicate fields by using only the first instance.

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

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

For example:

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

3.2 Negotiating a Connection

3.2.1 Goals and Principles

Negotiation means that one party in a discussion declares an intention or capability and the other party either acknowledges it, modifies it, or rejects it. In AMQP/Fast, we negotiate a number of specific aspects of the protocol:

1. The actual protocol and version.

2. Encryption arguments and the authentication of both parties.
3. Operational constraints.

3.2.2 Protocol and Version

An AMQP client and server agree on a protocol and version using this negotiation model:

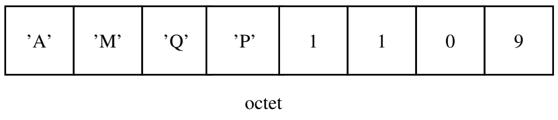
1. The client opens a new socket connection on a well-known or configured port and and sends an initiation sequence consisting of the text "AMQP" followed by a protocol ID (2 decimal digits) and a protocol version (2 decimal digits holding the high and low version numbers).
2. The server either accepts or rejects the initiation sequence. If it rejects the initiation sequence it closes the socket. Otherwise it leaves the socket open and implements the protocol accordingly.

Protocol grammar:

```
protocol-header     = 'AMQP' protocol-id protocol-version
protocol-id         = protocol-class protocol-instance
protocol-class      = OCTET
protocol-instance   = OCTET
protocol-version    = protocol-major protocol-minor
protocol-major      = OCTET
protocol-minor      = OCTET
```

For AMQP/Fast these are the correct values:

- protocol-class = 1 (this is the class of all AMQP/Fast protocols)
- protocol-instance = 1 (this is the instance of AMQP/Fast over TCP/IP)
- protocol-major = 0
- protocol-minor = 9



Guidelines for implementors:

- An AMQP server MAY accept multiple AMQP protocols including AMQP/Fast on the same socket.
- An AMQP server MAY accept non-AMQP protocols such as HTTP.
- An AMQP server MUST accept the AMQP/Fast protocol as defined by class = 1, instance = 1.
- If the server does not recognise the first 4 octets of data on the socket, or does not support the specific protocol version that the client requests, it MUST close the socket without sending any response back to the client.
- An AMQP client MAY detect the server protocol version by attempting to connect with its highest supported version and decreasing this if the server rejects the connection.
- An AMQP server MUST NOT ban or delay a client that requests an unsupported protocol version.
- An AMQP server MAY ban or delay a client that requests unknown protocol IDs.

The protocol negotiation model is compatible with existing protocols such as HTTP that initiate a connection with an constant text string, and with firewalls that sniff the start of a protocol in order to decide what rules to apply to it.

3.2.3 Encryption and Authentication

AMQP/Fast uses the SASL architecture for security. SASL encapsulates TLS, GSSAPI, Kerberos, and other encryption and authentication technologies.

Security is negotiated between server and client as follows:

1. The server sends a challenge to the client. The challenge lists the security mechanisms that the server supports.
2. The client selects a suitable security mechanism and responds to the server with relevant information for that security mechanism.
3. The server can send another challenge, and the client another response, until the SASL layer at each end has received enough information.
4. The server can now use the selected security mechanism and authenticated client identity to perform access controls on its data and services.

The "relevant information" is an opaque binary blob that is passed between the SASL layers embedded in the client and in the server.

SASL gives us the freedom to replace the security libraries with better (more secure or faster) technologies at a later date without modifying the protocol, the client, or the server implementations.

3.2.4 Client and Server Limits

The protocol allows the client and server to agree on limits to ensure operational stability. Limits allow both parties to pre-allocate key buffers, avoiding deadlocks. Every incoming frame either obeys the agreed limits, so is "safe", or exceeds them, in which case the other party is faulty and can be disconnected.

Rather than fix the limits in the protocol, we negotiate them. This lets us tune the protocol dynamically for different types of efficiency, e.g. minimal memory consumption vs. maximum performance.

Limits are negotiated to the lowest agreed value as follows:

1. The server tells the client what limits it proposes.
2. The client can respond to lower the limits for its connection.

3.2.5 Identification and Capabilities

During the negotiation of limits the peers MUST exchange this mandatory information about themselves:

- The specific content domains that they support (see later).

And the peers MAY exchange this optional information about themselves:

- Their public product name, for logging and tracking.
- Their version number.
- The platform they are running on.

For security reasons, all these pieces of information are optional and a highly-security conscious peer MAY choose to provide some or none of them. A peer MUST NOT use this information to adapt its behaviour.

3.3 Multiplexing and Pipelining

3.3.1 Goals and Principles

Connection multiplexing allows multiple threads of communication to share a single socket connection. This is valuable when threads are short-lived since the cost of opening and closing TCP/IP sockets can be relatively high.

Pipelining means that each peer can send frames asynchronously without waiting for the recipient to acknowledge each one. This is an important facility because it greatly improves performance.

3.3.2 Multiplexing Design

AMQP/Fast uses the concept of "channel" to carry bi-directional streams of communication between peers. AMQP/Fast provides methods to open, use, and close channels. Large messages are broken into smaller frames so that channels get fair (round-robin) use of the socket connection.

3.3.3 Pipelining Design

Our design is based on these principles:

1. Most operations succeed. We can thus optimise traffic in many cases by stating that "no response" means "successful".
2. Channels represent serial streams, where the order of methods and their acknowledgements is stable.
3. In general we want to send data as fast as possible in an asynchronous fashion. Where necessary, we can implement windowing and throttling at a higher level.

These principles provide the basis for a simple and efficient pipelining design based on "selective synchronisation". Some methods are explicitly synchronous - a specific request is always paired with a specific reply unless there is an error - and other methods are asynchronous unless the sender asks for a synchronisation response.

The recipient indicates an error by closing the channel. The close method includes the ID of the method that caused the error.

Methods on a specific channel are processed strictly in order. Thus the client can set "synchronisation points" by asking for a confirmation, and handle errors unambiguously (all methods up to but not including the failed method will have succeeded).

3.4 The Framing Design

3.4.1 Goals and Principles

Framing is the part of the protocol where we define how data is sent "to the wire". The framing design used in AMQP/Fast is designed to be compact, easy and very fast to parse, extensible, and robust.

The key parts of this design are:

- How the transport layer carries data.
- How we delimit frames on the connection.
- How we multiplex frames on the connection.
- How different types of frame carry data.
- How we handle out-of-band transport.

- How we support future protocols (e.g. IPv6).

3.4.2 The Transport Layer

We assume a reliable stream-oriented network transport layer (TCP/IP or equivalent). If an unreliable transport layer is used, we assume that the AMQP/Fast frames would be wrapped with additional traffic-control information such as windowing.

AMQP/Fast explicitly excludes any support for traffic control but does not disallow this to be implemented in an additional layer. We may at a future point provide protocol wrappers that wrap AMQP/Fast frames with traffic management to allow reliable transfer over multicast and point-to-point UDP.

3.4.3 Delimiting Frames

TCP/IP is a stream protocol, i.e. there is no in-built mechanism for delimiting frames. Existing protocols solve this in several different ways:

- Sending a single frame per connection. This is simple but adds considerable overhead to the protocol due to the opening and closing of connections.
- Adding frame delimiters to the stream. This is used in protocols such as SMTP but has the disadvantage that the stream must be parsed to find the delimiters. This makes implementations of the protocol slow.
- Counting the size of frames and sending the size in front of each frame. This is the fastest approach, and our choice.

Each frame is thus sent as a ”frame header” which contains the frame size, followed by a ”frame body” of the specified size (the header is not counted). Frames can carry methods and other data. To read a frame we use this logic:

1. Read and decode the frame header to get the size of the frame body.
2. If the frame body is larger than the agreed limit, close the connection with a suitable error reply code.
3. Otherwise, read the specified number of octets into a frame buffer.
4. Decode the frame buffer as needed.

To write a frame we use this logic:

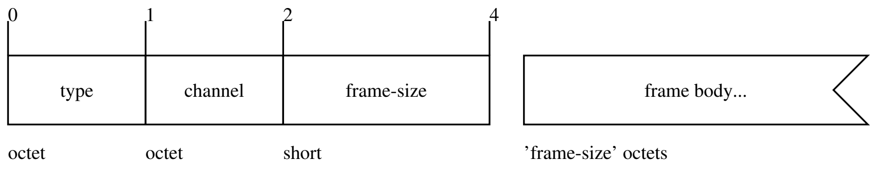
1. Marshal the frame buffer as needed.
2. Encode the length of this buffer into a frame header and write it.
3. Write the frame buffer contents, being the frame body.

Guidelines for implementors:

- A peer MUST write the frame body immediately after the frame header unless the body size is zero, or out-of-band transport is being used.

3.4.4 The Frame Header Wire Format

All frames start with a 4-octet header composed of a one-octet type indicator and a 3-octet size indicator:



AMQP/Fast defines several types of frame, with different type indicators:

- Type = 1, "METHOD": method frame.
- Type = 2, "HEADER": content header frame.
- Type = 3, "BODY": content body frame.
- Type = 4, "OOB METHOD": out-of-band method frame.
- Type = 5, "OOB HEADER": out-of-band band header frame.
- Type = 6, "OOB BODY": out-of-band body frame.
- Type = 7, "TRACE": trace frame.
- Type = 8, "HEARTBEAT": heartbeat frame.
- Type = 9, "META": meta frame.

The channel number is 0 for all frames which are global to the connection, and 1 .. 255 for frames that refer to specific channels (1 .. 64K-1 in extended frames).

Guidelines for implementors:

- If a peer receives a frame with a type that is not one of these defined types, it MUST treat this as a fatal protocol error and close the connection without sending any further data on it.

3.4.5 Method Frames

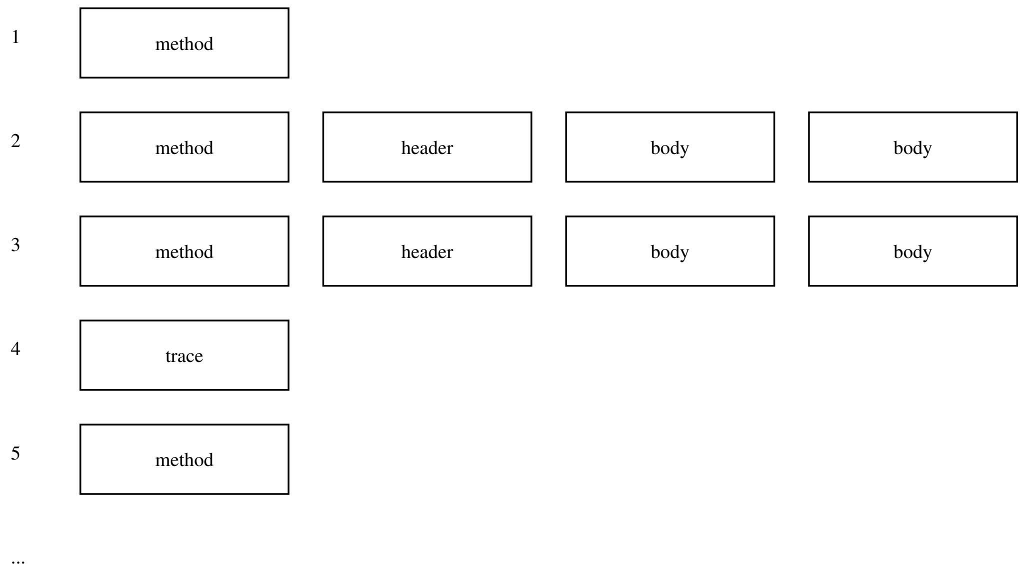
The method frame design is intended to be fast to read and write, and compact when only partly filled.

Method frames (also called "methods" in the following discussion, for brevity) are invariant. They contain no conditional or repeated fields. Methods are constructed out of a constant series of AMQP/Fast data fields (bits, integers, strings and string tables). Thus the marshalling code can be easily generated, and can be very rapid. In the OpenAMQ server and clients this code is generated directly from the protocol specifications.

3.4.6 Content Frames

Certain methods signal to the recipient that there is content arriving on the connection. Simple content consists of a header frame followed by zero or more body frames. Structured content may consist of a series of header frames followed by the appropriate body frames.

Looking at the frames as they pass on the wire, we might see something like this:



There is a specific reason for placing content body in distinct frames as compared to including the header and body in the method. We want to support ”zero copy” techniques in which content is never marshalled or encoded, and can be sent via out-of-band transport such as shared memory or remote DMA.

Guidelines for implementors:

- The channel number in content frames MUST NOT be zero.
- Content frames on a specific channel form an strict list. That is, they may be mixed with frames for different channels, but two contents may not be mixed or overlapped on a single channel.

3.4.7 Out-of-Band Transport Frames

Method, content header, and content body frames can be sent using out-of-band transport. The frame header is sent on the normal connection but the frame body is sent via another mechanism. The specific out-of-band transport used, and its configuration, is defined when a channel is opened.

3.4.8 Trace Frames

Trace frames are intended for a ”trace handler” embedded in the recipient. The significance and implementation of the trace handler is implementation-defined.

Guidelines for implementors:

- Both server and client MAY send trace frames at any point in the connection after protocol negotiation and before a Connection.Close method.
- If the recipient of a trace frame does not have a suitable trace handler, it MUST discard the trace frame without signalling any error or fault.
- The semantics and structure and of trace frames including the channel number are not formally defined by AMQP/Fast and implementations MUST NOT assume any interoperability with respect to trace frames unless and until formal standards are defined for these.

3.4.9 Heartbeat Frames

Although TCP/IP guarantees that data is not dropped or corrupted it can be slow to detect that a peer process has gone ”offline”. The heartbeat frame allows peers to detect network failure rapidly. This frame has no body.

The peers negotiate heartbeat parameters at the start of a connection.

Guidelines for implementors:

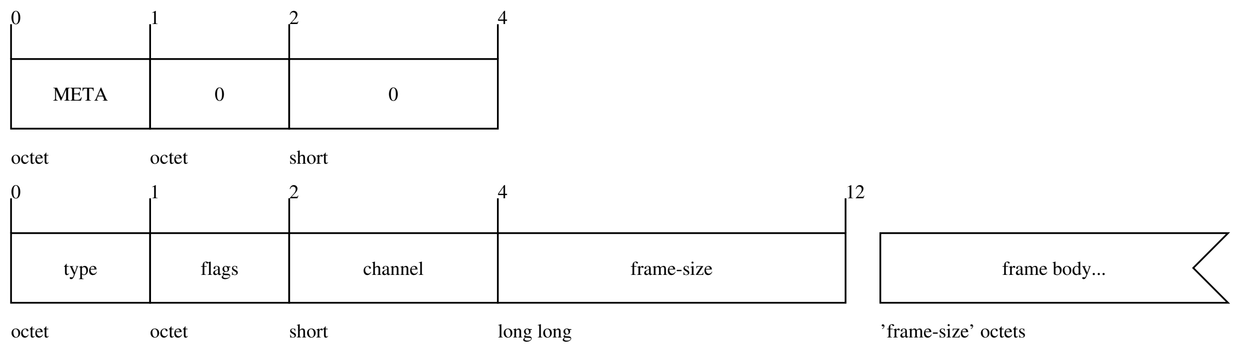
- The channel number in a heartbeat frame MUST be zero.
- An AMQP peer MUST handle a non-zero channel number as a connection exception.

3.4.10 Meta Frames

Meta frames are used to send frames of 64KB or larger (the size of a normal frame body is limited to 64KB-1 octets), and to use channel numbers above 255. Meta frames are designed to support high-capacity out-of-band transports and IPv6 (which has a 4GB frame limit, although ethernet still cannot handle frames larger than 12000 bytes).

3.4.11 Extended Frame Header Wire Format

A meta frame is followed by an ”extended frame header” with a short integer channel number and a long long size:



Guidelines for implementors:

- The meta frame channel number and size MUST be zero.
- The sender MUST NOT send intervening frames between the meta frame and the extended frame.
- After a meta frame all frame types are valid except a meta frame.
- The ”flags” octet is reserved for future use and MUST be be set to zero.
- The frame body is not affected in any way by the presence or absence of a meta frame, except that the frame body MAY be larger than 64KB after a meta frame.

3.5 The Abstract Content Model

AMQP/Fast uses an abstract content model that has these goals and features:

- It supports tree-structured content in which each content can contain further content (”child-content”), to any level.
- It provides multiple content types to allow optimal encoding for different applications.
- It supports content bodies of any size from zero octets upwards.
- Content body can be read and written directly from application memory with no formatting or copying (”zero copy”).
- Content headers are sent before content body so that the recipient can selectively discard content that it does not want to process.
- Content body is sent in separate frames to support the AMQP/Fast channel multiplexing model.

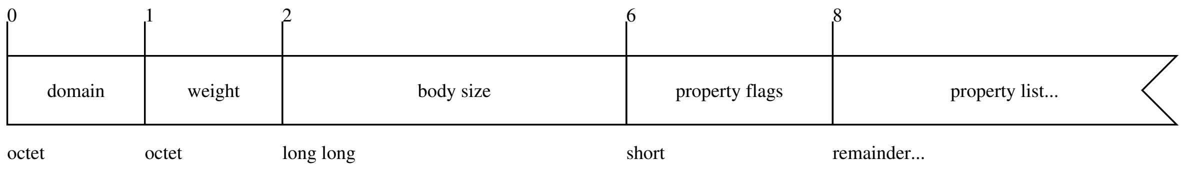
3.5.1 Formal Grammar for Content Frames

This is the formal grammar for the AMQP/Fast content model:

```
content         = header-frame child-content *body-frame
header-frame    = HEADER channel frame-size header-payload
channel         = octet
frame-size      = short-integer
header-payload  = class weight body-size
                  property-flags property-list
content-class   = OCTET
content-weight  = OCTET
body-size       = long-long-integer
property-flags  = 15*BIT %b0 / 15*BIT %b1 property-flags
property-list   = amqp-field
child-content   = weight*content
body-frame      = BODY channel frame-size body-payload
body-payload    = *OCTET
```

3.5.2 Content Header Frame Wire Format

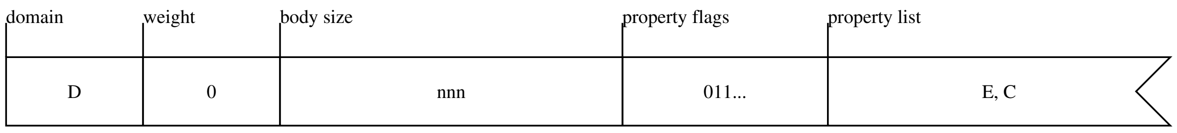
A content header frame has this format:



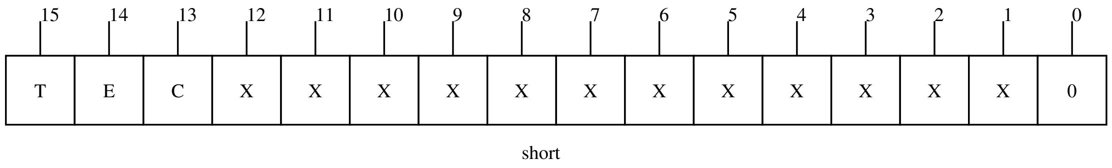
- The content class specifies the set of properties (with a predefined syntax and semantics) that the header frame may hold. The standard AMQP/Fast content classes are defined later.
- The weight field specifies whether the content is structured or not. Unstructured content has a weight of zero. Structured content has a weight of 1 to 255. This is the number of child-contents that the content contains.
- The body size is a 64-bit value that defines the total size of the body content. It may be zero, indicating that there will be no body frames.
- The property flags are a bit array that indicates the presence or absence of each property value.
- The property values are class-specific data fields. The properties are formatted as AMQP/Fast data types (integers, strings, field tables).

The set of properties for a content class can be any size. The first two properties for all content classes are the content MIME type and the content encoding. Following these, each content class has a specific set of properties defined in a strict order from most to least significant.

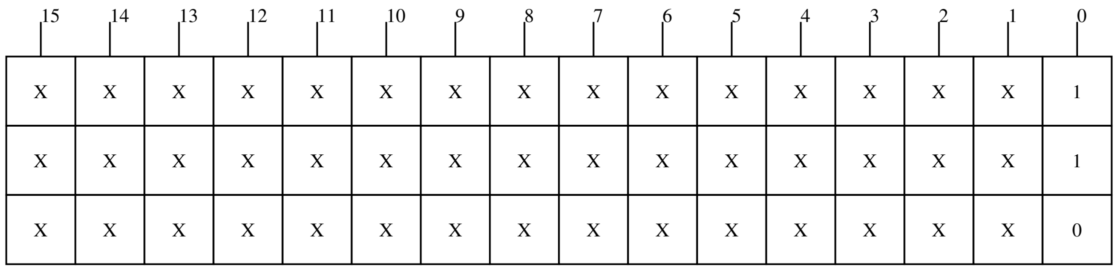
As an example, we take an imaginary content class "D" which has three properties, T, E and C. Imagine a simple content that has properties E and C but not T. The header frame will be formatted thus:



The property flags are ordered from high to low, the first property being indicated by bit 15, and so on:

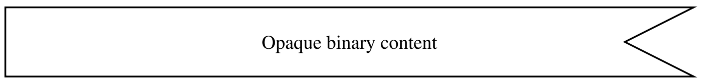


The property flags can specify more than 16 properties. If bit 0 is set, it indicates that a further property flags field follows. There can be several property flag fields in succession:


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X 1
X X X X X X X X X X X X X X X 1
X X X X X X X X X X X X X X X 0

3.5.3 Content Body Frame Wire Format

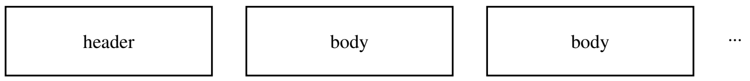
A content body frame has this format:



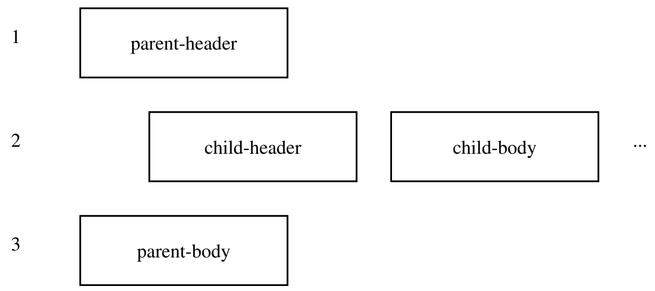
Where the size of the frame is defined in the frame header.

3.5.4 Structured Content

The "weight" field in the content header frame indicates whether the content has child-contents or not. In the simplest case (weight = zero) content consists of a header frame plus zero or more body frames:



If the content has child-contents, these are inserted between the header and body of the parent content:



This organisation means that a principal content (e.g. a very large video file) can have child content (e.g. subtitles and menus) which will be sent before the main content. If a more specific order of content is needed, it can be done by defining a top-level content that has no body, only a header.

3.5.5 Multiplexed Channel Content

Content consists of one or more frames. The recipient of content that has been sent in multiple frames can unambiguously reconstruct the content by implementing the content grammar defined above, on a per-channel basis.

Guidelines for implementors:

- Content frames MUST be sent on a single channel, and MUST NOT be intermixed with non-content frames on the same channel (method, trace, and heartbeat frames).
- Content frames MUST NOT be intermixed with frames of a different content.
- A peer MUST NOT make any assumptions about the minimum or maximum size of a body frame except that the minimum size is zero bytes and the maximum size is the smallest of either the negotiated maximum frame size or the remaining expected of content body (body-size minus amount of body data received so far).
- Empty body frames are permitted but a peer may not assign any special meaning to these: it MUST discard them with no side-effects of any kind.

3.5.6 Content Classes

The content class is not intended to provide data typing or encoding information. Its purpose is to allow functional clarity in the property sets and methods that we define per class of message.

Content classes turn AMQP/Fast from an abstract content-carrying protocol into a functional tool by providing data and operations that are directly and clearly mapped to the kinds of API that an application programmer needs to see.

AMQP/Fast defines these standard content classes:

- JMS: the content is a JMS-compliant message.
- File: the content is a file.
- Stream: the content is a streamed message.

Note that the ID of each content class is the same as the ID of the protocol class (JMS, File, Stream) that supports it.

Each class has a specific access model, though they all share the basic AMQP queueing and routing mechanisms including topic multipliers. These classes are defined in the next sections. This model lets us extend AMQP/Fast into new functional areas by defining new content classes and appropriate methods, without breaking interoperability.

Guidelines for implementors:

- The client MAY support any or all of the defined content classes.
- The server MUST support at least one of the defined content classes.

3.5.7 Content Type and Encoding

AMQP/Fast enforces MIME-compliance on all content classes using two standard properties that are the first defined in all content classes:

- ContentType (short string) - the MIME content type of the message body. The default value is "application/octet-stream".
- ContentEncoding (short string) - The content encoding of the message body. The default value is "binary".

Guidelines for implementors:

- The server SHOULD not modify the ContentType and ContentEncoding for content moved between producing and consuming clients.
- The server MAY set the ContentType and ContentEncoding properties for content produced by the server or by applications embedded in the server.

3.6 The Class/Method Model

3.6.1 Goals and Principles

AMQP/Fast allows two peers to connect and exchange data. There are several types of data and several ways of exchanging this data. The combinations rapidly get complex. Worse, it is not always clear which replies depend on which requests, since the exchange of data is often asynchronous, pipelined, and generally hard to follow.

Rather than attempt to formalise the exact flow of data from client to server and back, we use a metaphor taken from the software world, namely the object-oriented metaphor of "classes" and "methods".

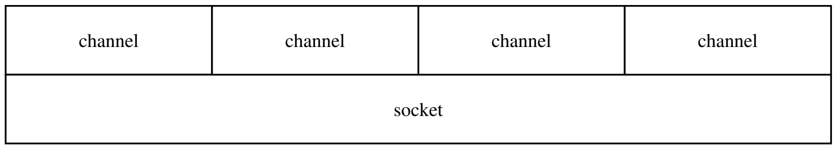
We build this as follows:

1. Each peer supports a set of classes. These classes have names and cover specific areas of functionality. A peer can support all the AMQP/Fast classes, or a subset of them. Since the operational functionality of the protocol depends on the classes, the operational functionality of a particular implementation is clear to see.
2. Each class supports a set of methods. These methods have names and provide specific functionalities. A peer can support all the AMQP/Fast methods in a particular class, or a subset of them. As with classes, this unambiguously defines the operational functionality of a peer.
3. The methods are either synchronous or asynchronous. A synchronous method replies immediately. An asynchronous method replies at some later time and may send multiple replies. A method "replies" by invoking the appropriate method in the requesting peer. A request and reply always involves two methods: one to implement the request and one to implement the reply.

4. Each method is either a client method, server method, or provided by both client and server). A particular method in a particular class always has an identical behaviour whether it is in a "client" or "server".
5. A peer can thus combine the roles of client and server in various degrees by implementing the necessary classes and methods. At any point the operational functionality of the peer can be unambiguously defined by the methods that it supports.

3.6.2 Client and Server Roles

The AMQP/Fast protocol is quasi-symmetrical with client-initiated connections. We can visualise the protocol as governing two levels of connection - network connections (sockets) and virtual connections (channels):



Each peer involved in a connection (socket and channel) is strictly a client or a server. The client initiates the connection and the server accepts the connection. This applies both to network and to virtual channel connections:

1. The client requests the server to open connection.
2. The server responds and the peers agree on connection parameters.
3. The client and the server exchange information.
4. Either peer requests to close the connection.
5. The other peer responds and they negotiate the shut-down.

AMQP/Fast forsees two main architectures that use this model:

1. Client-server, in which the same peer acts as client in all connections. This is the "usual" architecture.
2. Peer-to-peer, in which one peer acts as client in the network connection but the peers can take either role in the virtual connection.

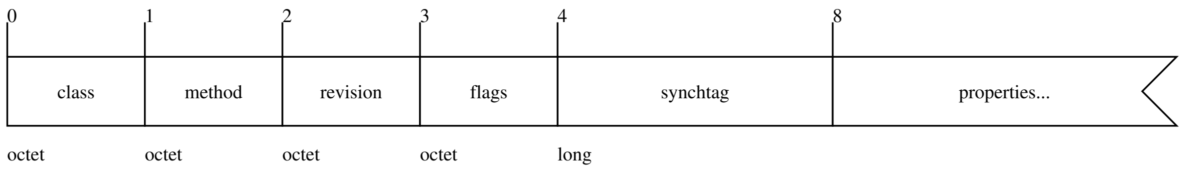
The peer-to-peer architecture requires that both peers can act as both client and server.

Guidelines for implementors:

- A peer MUST support at least either the client or server role.
- A peer MAY support both roles.

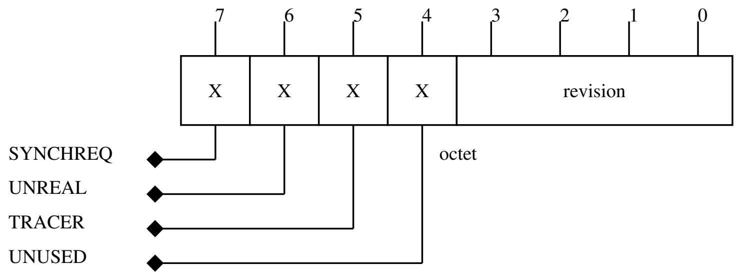
3.6.3 Method Frame Body Wire Format

A method frame body has this format:



- The class and method are protocol-constant values.
- The revision number indicates a revision of the method with new or modified properties.
- The flags provide a set of operational indicators.
- The synctag is an arbitrary value that the sender chooses to identify the method instance.
- The properties are a set of AMQP/Fast fields that specific to each method.

The flags are defined as follows:



- 7 = SYNCHREQ - if 1, requests a "synchpoint", turning an asynchronous method into a synchronous one. Has no effect on a synchronous method.
- 6 = UNREAL - if 1, requests an "unreal method", in which the recipient responds "as expected" but does no real work.
- 5 = TRACER - if 1, requests a "distributed trace", in which the recipient will trace all responses to this and other clients which the method provokes.
- 4 = UNUSED - this bit is not used.
- 3-0 = revision - defines the method revision.

Guidelines for implementors:

- The revision MUST be zero in major releases of the protocol. It is used to indicate revised methods in minor releases of the protocol. A peer MAY support multiple revisions of a protocol. A peer MUST support the specific method revisions defined in the protocol minor version number agreed during connection negotiation.
- synctags SHOULD BE unique since the last synchpoint (we explain synchpoints later).
- Bits 3 to 0 of the flags octet are reserved for future use and MUST be zero.

3.6.4 The Synchpoint Function

A client can request a synchpoint when sending an asynchronous request method. The server responds with a Channel.Synch method when it finishes processing the request method. Synchpoints are only sent if the method completes successfully. If the method fails, the server replies with a Channel.Close or Connection.Close depending on the severity of the error.

Clients can use synchpoints to force occasional synchronisation or to force a fully-synchronous dialogue. E.g. A client can be designed to use synchpoints at every asynchronous method, which turns the entire protocol into a synchronous protocol. This may be simpler for some scenarios than a normal asynchronous exchange of methods.

Guidelines for implementors:

- The server MUST ignore the SYNCHREQ flag on all methods that require a synchronous reply.

- The server MUST NOT set the SYNCHREQ flag on methods sent to the client, and the client MUST treat any method with the SYNCHREQ flag as a fatal connection error.

3.6.5 The Interface Test Function

A peer can test the capability of the other peer by sending methods with the UNREAL flag set. When a peer receives a message with this flag set it responds with a suitable ”as expected” response with arbitrary data and the UNREAL flag also set.

The goal of the UNREAL flag is to allow a monitor client to determine what methods a peer supports.

Guidelines for implementors:

- A peer SHOULD respect the UNREAL flag on all methods that it supports.
- A peer that supports the UNREAL flag MUST support unreal methods in any state.
- A peer SHOULD reply to unsupported methods with the UNREAL flag set, with a Connection.Unknown method.
- A peer MUST treat an unsupported methods with the UNREAL flag cleared as a fatal error and close the connection.
- A peer MUST respond to a supported method with the UNREAL flag set as follows: (a) decode the method and perform syntactic validation. (b) create a suitable response method with suitable arguments and send it with the UNREAL flag set. (c) not change any internal state.

3.6.6 The Distributed Trace Function

A distributed trace is a temporary and single-threaded trace of all messages produced by a specific server method. A distributed trace will last as long as the effects of the original method. There is no functionality to end a distributed trace, nor to distinguish more than one trace at once.

Guidelines for implementors:

- A server SHOULD implement the DTRACE flag as follows: (a) process the method as normal. (b) set the DTRACE flag on all methods, to all clients, that are the direct or indirect consequence of the original received method.

3.7 The Abstract Routing Model

AMQ defines an abstract model which all AMQP servers and clients must adhere to. The abstract routing model provides a formal and run-time configurable definition of how messages are stored and forwarded.

3.7.1 Introduction to the Abstract Routing Model

3.7.1.1 Restatement of Goals

AMQ is designed to carry data (”messages”) between applications (”clients”) written in arbitrary languages and running on arbitrary platforms spread across a fully-addressable network.

AMQ is designed to support many different types of application, each with its own requirements as to:

- Message size, from zero bytes up to many gigabytes.
- Number of clients, from one up to many thousands.

- Volume of messages crossing the network, from a few per day to hundreds of thousands per second.
- Critical latency of a message, from a millisecond to an hour.
- Reliability of delivery, from "as reliable as possible, whatever the cost" to "best-effort delivery".
- Use of network, memory, CPU, and disk resources.

The same AMQ architecture must be able to work efficiently on a small embedded message router, as well as on a supercluster with dozens of CPUs and tens of gigabytes of memory.

3.7.1.2 General Design Method

Our design method relies on these principles:

1. Give the functional architect (the person responsible for designing an AMQ network, using AMQ technology) as much control as possible over the choices and tradeoffs required for a specific functional scenario.
2. Design the architecture using abstract general models that the functional architect can combine as needed.

That is: rather than embed assumptions and decisions in the architecture we create abstract general-purpose constructs that can be reused in scenarios that we do not pretend to fully understand ourselves.

3.7.1.3 Main Insights

The main insights that led to our design (which is the third major refactoring of the AMQ protocol) were:

1. Subscriptions can be designed as a special case of queues. That is, subscriptions and queues can be designed to support exactly the same set of properties and access methods. In the topic/subscription model subscriptions are simply queues that have a special matching technique and a private name space.
2. Matching can be abstracted. That is, everything from matching on a specific destination name to matching on an SQL-like selector can be abstracted into a single concept.
3. Messages can have multiple instances. That is, a message can be in many queues at the same time. We do not copy message content, rather we use message reference counting. Queues store references to messages rather than messages. This applies both to persistent and non-persistent messages.
4. Queues and matching can be loosely-coupled. That is, while one client "writes to queue X" and a second client "reads from queue X", this can be done using a less tightly-coupled relationship than the obvious "producer writes to queue". Loose coupling is often desirable because it allows abstraction in which one side of a relationship can be reorganised without affecting the other.

We use these insights to define a set of building blocks that achieve much wider functionality than classic messaging middleware (like JMS) at no significant extra cost.

3.7.2 Overall Architecture

3.7.2.1 Classic Message-Oriented Middleware

A classic message-oriented middleware server provides two types of service: a "queue" service in which producers and consumers interact via shared named queues, and a "topic" service which allows clients to subscribe to a tree of topics. Clients subscribe with citeria such as the topic name, or a pattern representing a set of topics, e.g. "FOREX.*". The middleware explicitly provides functionality to subscribe, manage topics, and so on.

AMQ does not provide these services directly: the protocol and server make no mention of "topics" or "subscriptions". Rather, AMQ provides more general building blocks that can easily be used to build a topic service, a queue service, a mix of the two, and new kinds of service that fall outside the classic definitions.

3.7.2.2 Message, Queue, and Proxy

We define the "MESSAGE": a message is self-contained package of data that passes through the server. The server will not modify or separate the data in a message. The message is the basic unit of data transfer between clients and servers.

We define the "QUEUE": a queue holds messages and distributes these to clients. Queues are highly configurable objects that implement most of the quality of service aspects of an AMQP server. Queues can be private or shared, created by the server administrator, or by applications. Queues can holding their messages on disk or in memory, or a combination (working in memory and overflowing to disk when necessary). Queues can be durable (remaining active when the server stops and restarts) or transient (automatically deleted when no longer used).

Queues have public names, and when a client wants to get messages from a server it says, "give me messages from queue such-and-such".

We define the "PROXY": a proxy is a named object that accepts messages and distributes them to queues. A message can exist on many queues at once. Clients that want to send messages to a server always send them to a specific proxy.

3.7.2.3 Basic Architecture

The basic architecture is thus (showing a set of producer applications sending messages to a set of consumer applications):

```
                +---------+        +---------+
                +---------+        +---------+
                    |                   |
                    V                   V
      |         |     \               /      |
      |         |      +-------------+       |
      |         |      |   Proxy     |       |
      |         |      +-------------+       |
      |         |          /      \          |
      |         |  +--------+    +--------+  |
    __|__       |  | Queue  |    | Queue  |  |
    \ | /       |  +--------+    +--------+  |
     \|/        |     /               \      |
      V         +----/-----------------\-----+
                    |                   |
                    V                   V
                +---------+        +---------+
    Consumers:  | Client  |        | Client  |
                +---------+        +---------+
```

Note that:

1. The producers do not write directly to the consumers' queues.
2. The consumers do not read directly from the proxy.

3.7.2.4 Queue Consumers

While proxies copy messages across queues, queues distribute messages between consumers. A queue can have one consumer:

```
+-------+
| Queue |
+-------+
    |
    V
+--------+
| Client |
+--------+
```

A queue can also have many consumers:

```
                  +-------+
                  | Queue |
                  +-------+
                      |
      .----------.-----------.-----------.
      |          |           |           |
      V          V           V           V
  +--------+  +--------+  +--------+  +--------+
  | Client |  | Client |  | Client |  | Client |
  +--------+  +--------+  +--------+  +--------+
```

The queue distributes messages to the consumers according to a configurable traffic control model that is adjustable per consumer:

- Each consumer specifies whether it is going to acknowledge messages or not. Acknowledgements improve reliability but slow-down performance.
- Each consumer specifies how many messages the server should send it in advance of acknowledgements. This ”prefetch” window improves latency but can mean that consumers are not perfectly balanced.

Given these two parameters the queue will distribute messages to the consumers on a quasi-round-robin basis (i.e. to each consumer in turn, but without a guarantee of visting each consumer in any particular order).

3.7.2.5 Relationship between Queues and Proxies

A queue ”registers” with a proxy, asking: ”give me a copy of all messages that look like this”.

AMQ has a formal definition of a message which includes:

- the ”destination” for the message. This is a name that the client application uses. The destination name may (or may not) map directly to a queue name.
- the ”properties” for the message. These are a set of fields with a name, type, and value. The field properties may (or may not) be used by the proxy to decide which queues receive the messages.

3.7.3 The Proxy Design

3.7.3.1 Proxy Types

These are the basic set of proxies that an AMQP server MAY support:

1. ”fanout” routes all messages to all registered queues. Fanout proxies are used to create arbitrary routing flows within the server. For example, fanout proxies might be used to replicate data within a cluster.
2. ”dest-name” routes messages on their destination name using a direct comparison. That is, it passes messages to queues that have registered with ”give me all messages whose destination name is equal to this value”. Dest-name proxies are used to build conventional point-to-point store-and-forward mechanisms.

3. ”dest-wild” routes messages on their destination name using a wild card comparison. That is, it passes messages to queues that have registered with ”give me all messages whose destination name matches this pattern”. Dest-wild proxies are used to build conventional topic publish-and-subscribe mechanisms.
4. ”prop-test” routes messages on the value and presence of properties. That is, it passes messages to queues that have registered with ”give me all messages with properties with these names and values”. Prop-test proxies are used to build content-based routing mechanisms.
5. ”prop-wild” routes messages on a set of properties using wild card comparisons. That is, it passes messages to queues that have registered with ”give me all messages with properties that match these specifications”. Prop-wild proxies are used to build content-based routing mechanisms.
6. ”selector” routes messages using a programmable selector statement. That is, it passes messages to queues that have registered with ”give me all messages where this statement is true”. The selector statements are based on SQL WHERE conditions. Selector proxies are used to build content-based routing mechanisms.
7. ”lossy” routes messages with random failure. That is, it passes messages to all registered queues, but randomly drops messages. Lossy proxies are used to simulate network or processing failures.

Guidelines for implementors:

- An AMQP server MUST implement at least the fanout and dest-name proxies. An AMQP server MAY implement the other proxy types.
- An AMQP server MAY implement custom proxy types (not on this list). The name of a custom proxy type must start with ”x-”.

3.7.3.2 Proxy Implementation

Each proxy is a processing class in the server, with properties and methods that allow it to operate. All proxies have a common abstract structure:

```
+------+-------+-------------------+------------------------------+
| Name | Class | Registration list | Class-specific properties... |
+------+-------+-------------------+------------------------------+
                        :
                +--------------+    +-------+
                | Registration |--->| Queue |
                +--------------+    +-------+
                        :
                +--------------+    +-------+
                | Registration |--->| Queue |
                +--------------+    +-------+
                        :
                       ...
```

3.7.3.3 Multiple Registrations

A queue can register with the same proxy several times:

```
Register with dest-name: messages for destination X
Register with dest-name: messages for destination Y
Register with dest-name: messages for destination Z
```

The meaning of this is obvious and does not need much explaining: the registrations are acummulated so the queue will get messages for any of the three destination names:

```
+-----------+
| dest-name |
+-----------+
    | | |  "messages for destinations X, Y, Z"
  +-------+
  | Queue |
  +-------+
```

A queue can also register with two different proxies:

```
Register with dest-name: messages for destination X
Register with dest-name: messages for destination Y
Register with dest-name: messages for destination Z
Register with lossy
```

The meaning of this is as follows:

```
+-----------+
| dest-name |
+-----------+
    | | |  "messages for destinations X, Y, Z"
  +-------+
  | lossy |
  +-------+
      |    "all messages"
  +-------+
  | Queue |
  +-------+
```

In other words, the lossy proxy steps in between the queue and the dest-name proxy. We could imagine several levels:

```
+-----------+
| dest-name |
+-----------+
    | | |  "messages for destinations X, Y, Z"
+-----------+
| prop-name |
+-----------+
      |    "messages with property ABC"
  +-------+
  | lossy |
  +-------+
      |    "all messages"
  +-------+
  | Queue |
  +-------+
```

Guidelines for implementors:

- An AMQP server SHOULD support multiple registrations but MAY restrict its proxy implementation to a single proxy or single registration per queue.

3.7.3.4 On-Demand Wiring

To provide maximum flexibilty we split the definition of a server's wiring (i.e. the proxies, queues, and relationships between them) into three layers:

1. The set of proxy classes and how they are implemented. This layer is defined in the server's software.
2. A set of pre-defined proxies and queue templates. This layer is defined in the server's configuration files.
3. The final set of proxies, queues and registrations. This layer is defined at runtime by the application.

```
   +--------------------------+
1  |  Software constructions  |   E.g. what proxy classes exist?
   +--------------------------+
               |
               V
   +--------------------------+
2  |   Configured defaults    |   E.g. what queue templates exist?
   +--------------------------+
               |
               V
   +--------------------------+
3  |     On-demand wiring     |   E.g. create proxies & queues
   +--------------------------+
               |
               V
          Actual Wiring
```

Thus the protocol provides functionality to create and destroy proxies and queues, and register queues with proxies.

Guidelines for implementors:

- An AMQP server SHOULD support queue templating in which queues created at runtime can be configured by reference to a pre-configured template.

3.7.3.5 Durable Wiring

An AMQ server may hold purely transient data, but may equally hold persistent data. In the first case we want a ”clean restart” if a server is stopped and restarted. I.e. all dynamically-created proxy and queue definitions should be wiped. In the second case we want a ”reliable restart” so that queues keep their contents safely.

We use the concept of a ”durable resource”. This is a resource created by the client application but held by the server as if it was a configured resource. We need durable proxies, durable queues, and durable registrations. The opposite of ”durable” is ”transient”.

The actual or required durability of a resource is defined as follows:

1. If a resource is part of the server's configuration, it is always durable. For example, a proxy defined in the server configuration is always durable.
2. When a resource is created the client application may ask for it to be durable. For example, a queue can be created as ”durable”.
3. If the client application attempts to create a dependency from a durable resource onto a transient resource, this causes a channel exception. For example, if a client attempts to register a durable queue with a transient proxy the server will close the channel with an appropriate error.

3.7.4 Worked Examples

3.7.4.1 Reconstructing Store and Forward

Let us reconstruct queue-based ”store and forward” using these building blocks:

- We create a dest-name proxy called ”queue”.
- We agree that queue names are equal to the destination name placed in each message.
- We create a set of queues and register these with the ”queue” proxy.
- Publishers send their messages to the ”queue” proxy, specifying a destination name with each message.
- The ”queue” proxy does a lookup on the destination name and passes the message to the matching queue.
- Consumers get their messages from each named queue.

3.7.4.2 Reconstructing Topics and Subscriptions

Let us reconstruct topic-based ”subscribe and publish” using these building blocks:

- We create a dest-wild proxy called ”topic”.
- We agree that queue names are subscription names.
- We create one queue per subscription and attach these queues to the ”topic” proxy.
- Publishers send their messages to the ”topic” proxy, specifying a destination name with each message.

- The "topic" proxy does a topic-name match on the destination name and passes the message to all queues that match.
- Consumers get their messages from each named queue.

3.7.4.3 Reconstructing Temporary Destinations

- A temporary queue is simply a queue created by a client, and flagged for automatic deletion when the client disconnects.
- A temporary topic has no meaning: topics do not exist except as values on which to match.

3.7.5 Implementation Details

3.7.5.1 Queue Namespaces

We want to provide the client application with a consistent way to distinguish different types of queue, for example "subscriptions" and "private reply queues". These types have no meaning to the server but to avoid non-standard naming conventions we introduce the concept of a "queue namespace", which is a type name. The server simply prefixes the queue name with the namespace to create a unique internal name.

By convention in many scenarios, the namespace will be the same as the proxy name.

Thus a publisher may send to a proxy called "queue", while the consumer might read messages from a queue in namespace "queue".

3.7.5.2 Queue Templating and Tuning

Templating is a normalisation technique: rather than define many objects with similar properties we define a set of templates, and then refer to these when we define new objects. In object-oriented development this may be called a "class".

A queue template provides different types of information:

1. Organisation and tuning: file system criteria, maximum queue size, dead-letter queue, etc.
2. Constraints: e.g. max consumers.

Queue templating could be used to simplify the protocol. For example it is useful to be able to specify "this queue must be deleted when there are no more consumers". This could be a protocol flag or it could be a template property, understood by the server but not part of the explicit protocol.

We have to decide which queue properties are made visible in the protocol, and which can be configured in templates. Our choice is based on the requirement for interoperability:

- A property that has an impact on interoperability must be visible in the protocol.
- A property that must be tunable by the administrator must be part of the template.

Thus the "delete this queue when there are no more consumers" must be specified in the protocol, since it is necessary for interoperability. The fact that a queue automatically disappears when it is no longer needed is a vital part of some architectures. In contrast, a queue limit such as the maximum number of messages allowed in memory before the queue overflows onto disk is an adminisration concern, so must be part of templated configuration.

3.7.5.3 Registration Normalisation

In some scenarios many queues will register with the same proxies using identical registrations. We can foresee that this will create problems in high-volume scenarios - e.g. when several thousands of queues register with the same proxy.

Our class design normalises identical registrations, using a list of queues per registration:

```
+------+-------+-------------------+-----------------------------+
| Name | Class | Registration list | Class-specific properties... |
+------+-------+-------------------+-----------------------------+
                       :
                +--------------+    +-------+    +-------+
                | Registration |--->| Queue |--->| Queue |--->...
                +--------------+    +-------+    +-------+
                       :
                +--------------+    +-------+
                | Registration |--->| Queue |--->...
                +--------------+    +-------+
                       :
                      ...
```

3.8 Queueing and Routing Mechanisms

3.8.1 Goals and Principles

The AMQP queueing and routing mechanisms define how content is moved between an AMQP server and its clients. Our goal is to define a single coherent queueing and routing model that can support a range of different highly-functional content classes.

Our design is based on these elements:

- Virtual hosts: an independent collection of proxies and queues.
- Proxies: a proxy distributes messages across queues.
- Queues: a queue distributes messages between consumers.
- Message persistence: defining the level of assurance that a message will be successfully delivered despite crashes and failures.
- Message priorities: allowing messages to be delivered out-of-sequence according to their importance.
- Queue browsing: allowing a client to read messages in a simple synchronous manner.
- Acknowledgements: allowing a client to tell the server when it has successfully processed a message.
- Window-based flow control: allowing a client to control how many messages it receives
- Transactions: allowing a client to group multiple messages and acknowledgements into a single unit of work.

In general all these mechanisms apply to all content classes. Certain content classes may elect to use a simpler model for performance reasons. This is specifically the case for streamed content, which does not support persistence, browsing, acknowledgements, or transactions.

3.8.2 Virtual Hosts

AMQP/Fast has explicit support for virtual hosts. The client chooses a virtual host as the last stage in negotiating the connection. A virtual host has its own set of proxies, queues and access controls.

3.8.3 Proxies

Topics are named objects that act as message filters and multipliers. Proxies do not hold messages: rather they copy messages to queues. All messages are published to proxies.

Guidelines for implementors:

- Proxies are not tied to specific content classes.
- A server MUST allow any content class to be sent to any proxy, in any mix, depending on the access rights of the producing client.
- A server SHOULD discard messages for which there are no valid queues. It MAY log such messages and/or send them to a dead-letter queue.

3.8.4 Queues

A queue is a named object that acts as an asynchronous buffer for messages. Queues may be held in memory, on disk, or some combination. Each virtual host provides a namespace for queues. Queues hold messages and distribute them between one or more clients (consumers). A message published to a queue is never sent to more than one client unless it is is being resent after a failure or rejection.

Note that mixed content on a queue is generally sent asynchronously using the appropriate Deliver methods, to those clients that have asked for it. The Browse method allows clients to selectively receive single JMS or File messages but not Stream messages.

Guidelines for implementors:

- Queues are not tied to specific content classes.
- A server MUST allow any content class to be sent to any queue, in any mix, depending on the access rights of the producing client.

3.8.5 Message Persistence

A persistent message is held securely on disk and guaranteed to be delivered even if there is a serious network failure, server crash, overflow etc.

Messages may be persistent or not, depending on several criteria:

1. In the JMS content class, messages are individually marked as persistent or non-persistent.
2. In the File content class, messages are always persistent.
3. In the Stream content class, messages are never persistent.

Guidelines for implementors:

- The server MUST do a best-effort to hold persistent messages on a fully-reliable storage medium.
- The server MUST NOT discard persistent messages in case of overflow but SHOULD use the Channel.Flow method to slow or stop a producer when necessary.
- The server MAY overflow non-persistent JMS or Stream messages to persistent storage, or MAY discard non-persistent JMS or Stream messages on a priority basis if the queue exceeds some configured maximum size.

3.8.6 Message Priorities

A high priority message is sent ahead of lower priority messages waiting in the same queue or subscription. When messages must be discarded in order to maintain a specific service quality level the server will first discard low-priority messages.

Guidelines for implementors:

- A server MUST implement at least 2 priority levels for non-persistent messages. A server MAY implement up to 10 priority levels.
- A server MAY implement all persistent messages as a single priority.

3.8.7 Acknowledgements

When the client finishes processing a message it tells the server via an acknowledgement. AMQP/Fast defines different acknowledgement models for different levels of performance and reliability:

- Acknowledgement by the application after it has processed a message.
- Acknowledgement by the client layer when it has received a message and before it passes it to the application.
- Acknowledgement by the server when it has sent a message to the client.

The first two models use the same protocol mechanics. The difference is when the client API sends the acknowledgement. The last model uses a specific protocol option, in the CONSUME method for those classes that support acknowledgement.

3.8.8 Flow Control

AMQP/Fast flow control is based on these concepts:

1. Allowing the client to specify a prefetch - how many messages it will receive without acknowledgement. We call this "window-based flow control".
2. Allowing one peer to halt and restart a flow of messages coming from the other peer, using the Channel.Flow method, which we call "channel flow control".

Since Publish methods are not acknowledged, there is no window-based flow control for publishing and the recommended way to pause a publisher is to use channel flow control.

Guidelines for implementors:

- A server MUST support window-based flow control for clients.
- A server MAY use channel flow control to pause publishers when queues or subscriptions overflow.
- A client MUST respect channel flow control when publishing messages to the server.
- A server MAY disconnect and/or ban clients that do not respect channel flow control methods.

3.8.9 Transactions

AMQP/Fast supports three kinds of transactions, each kind progressively slower, more complex, and more reliable:

1. Automatic transactions, in which every method is wrapped in a transaction. Automatic transactions provide no way to create units of work. Every message and acknowledgement is processed as a standalone transaction.

2. AMQP/Fast transactions, which cover units of work on a per-channel basis. Transactions cover messages published, and messages received and acknowledged. There is no "start transaction" method: a transaction starts when the channel is opened and after every commit or rollback. If the channel is closed without a commit, all pending work is rolled-back. Nested transactions are not allowed.
3. XA 2-phase transactions, using the XA class. This is still under construction.

The transaction type is chosen on a per-channel basis.

3.9 Summary of Classes and Methods

3.9.1 Goals and Principles

We aim to provide a clear and extensible organisation of the classes and methods that a peer provides. Classes and methods are numbered but also have constant names (shown in uppercase). The names are chosen to be consistent across classes, so a similar method in different classes has the same name and may have similar arguments where appropriate. The full definition of each class and method is provided later.

We define methods as being either:

- a synchronous request ("syn request"). The sending peer SHOULD wait for the specific reply method, but MAY implement this asynchronously.
- a synchronous reply ("syn reply for XYZ").
- an asynchronous request or reply ("async"). The sending peer will usually not expect a synchronous reply but MAY set the SYNCHREQ flag to request a synchpoint.

A full explanation of classes and methods is provided in the Reference section of this document.

3.9.2 Overall Grammar

```
connection          = open-connection *use-connection close-connection
open-connection     = C:protocol-header
                      S:START C:START-OK
                      *challenge
                      S:TUNE C:TUNE-OK
                      C:OPEN S:OPEN-OK
challenge           = S:SECURE C:SECURE-OK
use-connection      = S:UNKNOWN
                    / C:UNKNOWN
                    / channel
close-connection    = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
channel             = open-channel *use-channel close-channel
open-channel        = C:OPEN S:OPEN-OK
use-channel         = C:FLOW
                    / S:FLOW
                    / S:SYNCH
                    / access
                    / proxy
                    / queue
                    / jms
                    / file
                    / stream
                    / tx
                    / dtx
                    / test
close-channel       = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
access              = C:GRANT S:GRANT-OK
proxy               = C:ASSERT   S:ASSERT-OK
                    / C:DELETE   S:DELETE-OK
queue               = C:ASSERT   S:ASSERT-OK
                    / C:REGISTER S:REGISTER-OK
                    / C:CANCEL   S:CANCEL-OK
                    / C:PURGE    S:PURGE-OK
                    / C:DELETE   S:DELETE-OK
jms                 = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:PUBLISH content
                    / S:DELIVER content
                    / C:BROWSE ( S:BROWSE-OK content / S:BROWSE_EMPTY )
                    / C:ACK
                    / C:REJECT
file                = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:OPEN S:OPEN-OK C:STAGE content
                    / S:OPEN C:OPEN-OK S:STAGE
                    / C:PUBLISH
                    / S:DELIVER
                    / C:ACK
                    / C:REJECT
stream              = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:PUBLISH content
                    / S:DELIVER content
tx                  = C:COMMIT S:COMMIT-OK
                    / C:ABORT S:ABORT-OK
dtx                 = C:START S:START-OK
test                = C:INTEGER S:INTEGER-OK
                    / S:INTEGER C:INTEGER-OK
                    / C:STRING S:STRING-OK
                    / S:STRING C:STRING-OK
                    / C:TABLE S:TABLE-OK
                    / S:TABLE C:TABLE-OK
                    / C:CONTENT S:CONTENT-OK
                    / S:CONTENT C:CONTENT-OK
```

3.9.3 Connection Class - Work With Socket Connections

Connection.Start ID=1/1 - client, sync request, start connection negotiation.

```
version_major      octet       #  negotiated protocol major version
version_minor      octet       #  negotiated protocol major version
mechanisms         longstr     #  available security mechanisms
locales            longstr     #  available message locales
```

Connection.Start-Ok ID=1/2 - server, sync reply for Start, select security mechanism and locale.

```
mechanism          shortstr    #  selected security mechanism
locale             shortstr    #  selected message locale
```

Connection.Secure ID=1/3 - client, sync request, seurity mechanism challenge.

```
challenge          longstr     #  security challenge data
```

Connection.Secure-Ok ID=1/4 - server, sync reply for Secure, seurity mechanism response.

```
response           longstr     #  security response data
```

Connection.Tune ID=1/5 - client, sync request, propose connection tuning parameters.

```
frame_max          long        #  maximum frame size
channel_max        short       #  maximum number of channels
access_max         short       #  maximum number of access tickets
consumer_max       short       #  maximum consumers per channel
heartbeat          short       #  desired heartbeat delay
txn_limit          short       #  maximum transaction size
jms_support        bit         #  JMS content is supported?
file_support       bit         #  file content is supported?
stream_support     bit         #  stream content is supported?
```

Connection.Tune-Ok ID=1/6 - server, sync reply for Tune, negotiate connection tuning parameters.

```
frame_max          long        #  maximum frame size
channel_max        short       #  maximum number of channels
ticket_max         short       #  maximum number of access tickets
heartbeat          short       #  desired heartbeat delay
jms_support        bit         #  JMS content is supported?
file_support       bit         #  file content is supported?
stream_support     bit         #  stream content is supported?
prefetch_max       long        #  maximum prefetch size for connection
```

Connection.Open ID=1/7 - server, sync request, open a path to a virtual host.

```
virtual_path       shortstr    #  path value virtual server path
client_id          shortstr    #  client identifier
```

Connection.Open-Ok ID=1/8 - client, sync reply for Open, signal that the connection is ready.

```
client_id          shortstr    #  assigned client identifier
```

Connection.Unknown ID=1/9 - client, async, signal that an interface test method has failed.

```
class              octet       #  failing method class
method             octet       #  failing method ID
synchtag           short       #  failing method synchtag
reply_code         short       #  reply code from server reply code
reply_text         shortstr    #  localised reply text localised reply text
```

Connection.Close ID=1/10 - client, sync request, request a connection close.

```
reply_code         short       #  reply code from server reply code
reply_text         shortstr    #  localised reply text localised reply text
class              octet       #  failing method class
method             octet       #  failing method ID
synchtag           short       #  failing method synchtag
```

Connection.Close-Ok ID=1/11 - client, sync reply for Close, confirm a connection close. No specific fields.

3.9.4 Channel Class - Work With Channels

Channel.Open ID=2/1 - server, sync request, open a channel for use.

```
prefetch_max       long        #  maximum prefetch size for channel
out_of_band        shortstr    #  out-of-band settings for channel
tx_mode            octet       #  transaction mode for channel
```

Channel.Open-Ok ID=2/2 - client, sync reply for Open, signal that the channel is ready. No specific fields.

Channel.Flow ID=2/3 - client, async, enable/disable flow from peer.

```
flow_pause         bit         #  start/stop content frames
```

Channel.Close ID=2/4 - client, sync request, request a channel close.

```
reply_code         short       #  reply code from server reply code
reply_text         shortstr    #  localised reply text localised reply text
class              octet       #  failing method class
method             octet       #  failing method ID
synchtag           short       #  failing method synchtag
```

Channel.Close-Ok ID=2/5 - client, sync reply for Close, confirm a channel close. No specific fields.

3.9.5 Access Class - Work With Access Tickets

Access.Grant ID=3/1 - server, sync request, request an access ticket.

```
realm              shortstr    #  path value realm to work with
exclusive          bit         #  request exclusive access
passive            bit         #  request passive access
active             bit         #  request active access
write              bit         #  request write access
read               bit         #  request read access
```

Access.Grant-Ok ID=3/2 - client, sync reply for Grant, grant access to server resources.

```
ticket             short       #  access ticket granted by server
```

3.9.6 Proxy Class - Work With Proxies

Proxy.Assert ID=4/1 - server, sync request, create proxy if needed.

```
ticket             short       #  access ticket granted by server
proxy              shortstr    #  proxy name
class              shortstr    #  proxy class
if_exists          bit         #  do not create proxy
durable            bit         #  request a durable proxy
auto_delete        bit         #  auto-delete proxy when unused
```

Proxy.Assert-Ok ID=4/2 - client, sync reply for Assert, confirms a proxy definition. No specific fields.

Proxy.Delete ID=4/3 - server, sync request, delete a proxy.

```
ticket             short       #  access ticket granted by server
proxy              shortstr    #  proxy name
if_unused          bit         #  delete only if unused
```

Proxy.Delete-Ok ID=4/4 - client, sync reply for Delete, confirm deletion of a proxy. No specific fields.

3.9.7 Queue Class - Work With Queues

Queue.Assert ID=5/1 - server, sync request, create queue if needed.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
template           shortstr    #  queue template
if_exists          bit         #  do not create queue
durable            bit         #  request a durable queue
private            bit         #  request a private queue
auto_delete        bit         #  auto-delete queue when unused
```

Queue.Assert-Ok ID=5/2 - client, sync reply for Assert, confirms a queue definition.

```
queue              shortstr    #  queue name name of queue
proxy              shortstr    #  proxy name queue proxy, if any
message_count      long        #  number of messages in queue queue
consumer_count     long        #  number of consumers
```

Queue.Register ID=5/3 - server, sync request, register queue with a proxy.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
proxy              shortstr    #  proxy name proxy to register with
arguments          table       #  arguments for registration
```

Queue.Register-Ok ID=5/4 - client, sync reply for Register, confirm registration successful. No specific fields.

Queue.Purge ID=5/5 - server, sync request, purge a queue.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
```

Queue.Purge-Ok ID=5/6 - client, sync reply for Purge, confirms a queue purge.

```
message_count      long        #  number of messages purged
```

Queue.Delete ID=5/7 - server, sync request, delete a queue.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
if_unused          bit         #  delete only if unused
if_empty           bit         #  delete only if empty
```

Queue.Delete-Ok ID=5/8 - client, sync reply for Delete, confirm deletion of a queue.

```
message_count      long        #  number of messages purged
```

3.9.8 Jms Class - Work With Jms Content

Jms.Consume ID=6/1 - server, sync request, start a queue consumer.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
prefetch_size      short       #  prefetch window in octets
prefetch_count     short       #  prefetch window in messages
no_local           bit         #  do not receive own messages
auto_ack           bit         #  no acknowledgement needed
exclusive          bit         #  request exclusive access
```

Jms.Consume-Ok ID=6/2 - client, sync reply for Consume, confirm a new consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

Jms.Cancel ID=6/3 - server, sync request, end a queue consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

Jms.Cancel-Ok ID=6/4 - client, sync reply for Cancel, confirm a cancelled consumer. No specific fields.

Jms.Publish ID=6/5 - server, async, carries content, publish a message.

```
ticket             short       #  access ticket granted by server
proxy              shortstr    #  proxy name
immediate          bit         #  assert immediate delivery
```

Jms.Deliver ID=6/6 - client, async, carries content, notify the client of a consumer message.

```
delivery_tag       longlong    #  server-assigned delivery tag
redelivered        bit         #  signal a redelivered message
proxy              shortstr    #  proxy name
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
```

Jms.Browse ID=6/7 - server, sync request, direct access to a queue.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
no_local           bit         #  do not receive own messages
auto_ack           bit         #  no acknowledgement needed
```

Jms.Browse-Ok ID=6/8 - client, sync reply for Browse, carries content, provide client with a browsed message.

```
delivery_tag       longlong    #  server-assigned delivery tag
redelivered        bit         #  signal a redelivered message
message_count      long        #  number of messages pending
```

Jms.Browse Empty ID=6/9 - client, sync reply for Browse, indicate no messages available. No specific fields.

Jms.Ack ID=6/10 - server, async, acknowledge one or more messages.

```
delivery_tag       longlong    #  server-assigned delivery tag
multiple           bit         #  acknowledge multiple messages
```

Jms.Reject ID=6/11 - server, async, reject an incoming message.

```
delivery_tag       longlong    #  server-assigned delivery tag
requeue            bit         #  requeue the message
```

3.9.9 File Class - Work With File Content

File.Consume ID=7/1 - server, sync request, start a queue consumer.

```
ticket             short       #  access ticket granted by server
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
prefetch_size      short       #  prefetch window in octets
prefetch_count     short       #  prefetch window in messages
no_local           bit         #  do not receive own messages
auto_ack           bit         #  no acknowledgement needed
exclusive          bit         #  request exclusive access
```

File.Consume-Ok ID=7/2 - client, sync reply for Consume, confirm a new consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

File.Cancel ID=7/3 - server, sync request, end a queue consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

File.Cancel-Ok ID=7/4 - client, sync reply for Cancel, confirm a cancelled consumer. No specific fields.

File.Open ID=7/5 - client, sync request, request to start staging.

```
identifier         shortstr    #  staging identifier
content_size       longlong    #  message content size
```

File.Open-Ok ID=7/6 - client, sync request, confirm staging ready.

```
staged_size        longlong    #  already staged amount
```

File.Stage ID=7/7 - client, sync reply for Open-Ok, carries content, stage message content. No specific fields.

File.Publish ID=7/8 - server, async, publish a message.

```
ticket             short       #  access ticket granted by server
proxy              shortstr    #  proxy name
immediate          bit         #  assert immediate delivery
identifier         shortstr    #  staging identifier
```

File.Deliver ID=7/9 - client, async, notify the client of a consumer message.

```
delivery_tag       longlong    #  server-assigned delivery tag
redelivered        bit         #  signal a redelivered message
proxy              shortstr    #  proxy name
namespace          shortstr    #  queue namespace
queue              shortstr    #  queue name
identifier         shortstr    #  staging identifier
```

File.Ack ID=7/10 - server, async, acknowledge one or more messages.

```
delivery_tag       longlong    #  server-assigned delivery tag
multiple           bit         #  acknowledge multiple messages
```

File.Reject ID=7/11 - server, async, reject an incoming message.

```
delivery_tag       longlong    #  server-assigned delivery tag
requeue            bit         #  requeue the message
```

3.9.10 Stream Class - Work With Streaming Content

Stream.Consume ID=8/1 - server, sync request, start a queue consumer.

```
ticket            short      #  access ticket granted by server
namespace         shortstr   #  queue namespace
queue             shortstr   #  queue name
prefetch_size     short      #  prefetch window in octets
prefetch_count    short      #  prefetch window in messages
consume_rate      long       #  transfer rate in octets/second
no_local          bit        #  do not receive own messages
exclusive         bit        #  request exclusive access
```

Stream.Consume-Ok ID=8/2 - client, sync reply for Consume, confirm a new consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Stream.Cancel ID=8/3 - server, sync request, end a queue consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Stream.Cancel-Ok ID=8/4 - client, sync reply for Cancel, confirm a cancelled consumer. No specific fields.

Stream.Publish ID=8/5 - server, async, carries content, publish a message.

```
ticket            short      #  access ticket granted by server
proxy             shortstr   #  proxy name
immediate         bit        #  assert immediate delivery
```

Stream.Deliver ID=8/6 - client, async, carries content, notify the client of a consumer message.

```
delivery_tag      longlong   #  server-assigned delivery tag
redelivered       bit        #  signal a redelivered message
proxy             shortstr   #  proxy name
namespace         shortstr   #  queue namespace
queue             shortstr   #  queue name
```

3.9.11 Tx Class - Work With Standard Transactions

Tx.Commit ID=9/1 - server, sync request, commit the current transaction. No specific fields.

Tx.Commit-Ok ID=9/2 - client, sync reply for Commit, confirm a successful commit. No specific fields.

Tx.Abort ID=9/3 - server, sync request, abandon the current transaction. No specific fields.

Tx.Abort-Ok ID=9/4 - client, sync reply for Abort, confirm a successful abort. No specific fields.

3.9.12 Dtx Class - Work With Distributed Transactions

Dtx.Start ID=10/1 - server, sync request, start a new distributed transaction.

```
dtx_identifier    shortstr   #  distributed transaction identifier
```

Dtx.Start-Ok ID=10/2 - client, sync reply for Start, confirm the start of a new distributed transaction. No specific fields.

3.9.13 Test Class - Test Functional Primitives Of The Implementation

Test.Integer ID=11/1 - client, sync request, test integer handling.

```
integer_1         octet      #  octet test value
integer_2         short      #  short test value
integer_3         long       #  long test value
integer_4         longlong   #  long-long test value
operation         octet      #  operation to test
```

Test.Integer-Ok ID=11/2 - client, sync reply for Integer, report integer test result.

```
result             longlong    #  result value
```

Test.String ID=11/3 - client, sync request, test string handling.

```
string_1           shortstr    #  short string test value
string_2           longstr     #  long string test value
operation          octet       #  operation to test
```

Test.String-Ok ID=11/4 - client, sync reply for String, report string test result.

```
result             longstr     #  result value
```

Test.Table ID=11/5 - client, sync request, test field table handling.

```
table              table       #  field table of test values
integer_op         octet       #  operation to test on integers
string_op          octet       #  operation to test on strings
```

Test.Table-Ok ID=11/6 - client, sync reply for Table, report table test result.

```
integer_result     longlong    #  integer result value
string_result      longstr     #  string result value
```

Test.Content ID=11/7 - client, sync request, carries content, test content handling. No specific fields.

Test.Content-Ok ID=11/8 - client, sync reply for Content, carries content, report content test result.

```
content_checksum   long        #  content hash
```

3.9.14 Explanatory Notes

Methods are numbered locally per class and there is no attempt to keep a consistent numbering for similar methods in different classes, nor to distinguish client or server methods through special numbering schemes. Method numbering has no significance beyond uniquely identifying the method.

The grammars use this notation:

- 'S:' indicates data or a method sent from the server to the client.
- 'C:' indicates data or a method sent from the client to the server.
- +term or +(...) expression means '1 or more instances'.
- *term or *(...) expression means 'zero or more instances'.
- Methods are indicated by uppercase names, e.g. OPEN.

3.10 Error Handling

3.10.1 Goals and Principles

Error handling is a critical aspect of any protocol. First, we need a clear statement of what situations can provoke an error. Secondly, we need a clear way of reporting errors. Lastly we need unambiguous error handling that leaves both peers in a clear state.

The AMQP/Fast error handling model is based on an answer to each of these questions:

1. Use an exception-based model to define protocol correctness.
2. Use existing standards for error reporting.
3. Use a hand-shaked close to handle errors.

3.10.2 Existing Standards

The standard for error handling (defined semi-independently in several IETF RFCs) is the 3-digit reply code. This format has evolved into a fine-grained tool for communicating success or failure. It is also well-structured for expansion as a protocol gets more mature.

The current AMQP reply codes are standard to all protocols in the AMQP family and are defined in AMQ RFC 011.

The reply code is constructed as follows:

- The first digit - the completion indicator - reports whether the request succeeded or not.
- The second digit - the category indicator - provides more information on failures.
- The third digit - the instance indicator - distinguishes different situations with the same completion/category.

The completion indicator has one of these values:

```
1 = ready to be performed, pending some confirmation.
2 = successful.
3 = ready to be performed, pending more information.
4 = failed, but may succeed later.
5 = failed, requires intervention.
```

The category indicator has one of these values:

```
0 = error in syntax.
1 = the reply provides general information.
2 = problem with session or connection.
3 = problem with security.
4 = application-specific.
```

The instance indicator is 0 to 9 as needed to distinguish different situations.

3.10.3 The Assertion/Exception Model

AMQP/Fast uses an assertion/exception model that has these goals:

- identify and document all protocol preconditions ("assertions").
- define the exception level caused by any assertion failure.
- define a formal procedure for handling such exceptions.

AMQP/Fast defines two exception levels:

1. Channel exceptions. These close a single virtual connection. A channel exception is raised when a peer cannot complete some request because of transient or configuration errors.
2. Connection exceptions. These close the socket connection. A connection exception is raised when a peer detects a syntax error, badly-formed frame, or other indicator that the other peer is not conformant with AMQP/Fast.

We document the assertions formally in the definition of each class and method.

3.10.4 Hand-shaked Closure

Closing a connection for any reason - normal or exceptional - must be done carefully. Abrupt closure is not always detected rapidly, and in the case of errors, it means that error responses can be lost. The correct design is to hand-shake all closure so that we close only after we are sure the other party is aware of the situation.

A peer can close a channel or connection at any time for internal reasons, or as a reaction to an error. It sends a Close method to the other party. The receiving peer must respond to a Close with a Close-Ok method.

The closing peer reads methods back until it gets a Close-Ok, at which point it closes the connection and frees resources.

3.11 The JMS Operational Model

3.11.1 Goals and Principles

JMS is a standard API for messaging middleware servers. The AMQP/Fast JMS implementation is compatible with providers that conform to the SUN JMS specifications (within the bounds of JMS standardisation, which does not guarantee interoperability). JMS messages can be published and consumed using a set of methods that map to the JMS API, including:

- Publishing a message to a ”queue”.
- Publishing a message to a ”topic”.
- Creating a consumer for a queue.
- ”Subscribing” to a ”topic”.
- Browsing a destination for messages.

3.11.2 JMS Content Properties

These are the properties defined for JMS content:

- ContentType (short string)
- ContentEncoding (short string)
- DeliveryMode (octet)
- Priority (octet)
- CorrelationID (short string)
- ReplyTo (short string)
- Destination (short string)
- Expiration (short string)
- MessageID (short string)
- Timestamp (short string)
- Type (short string)
- UserID (short string)
- AppID (short string)
- Headers (field table)

3.12 The File Transfer Operational Model

3.12.1 Goals and Principles

The file transfer operational model is designed for enterprise-wide file transfer based on the AMQP/Fast queueing and routing models. For instance, routing files via the publish and subscribe functionality of topics and subscriptions.

File content is specifically different from JMS content in that:

- Files are always persistent.
- File transfer is restartable: if a file has been partially transferred when a connection is broken, the sender can resend just the remainder.

File transfers always happen in two steps:

1. The sender ”stages” the file into a temporary area provided by the recipient.
2. The sender notifies the recipient of the file, using Publish (a client to a server), or Deliver (a server to a client).

A File message is persistent and optimised for restartable transfers across possibly unreliable network connections.

3.12.2 File Content Properties

These are the properties defined for File content:

- ContentType (short string)
- ContentEncoding (short string)
- Priority (octet)
- ReplyTo (short string)
- Destination (short string)
- MessageID (short string)
- FileName (short string)
- RevisedTime (time stamp)
- Headers (field table)

3.13 The Stream Operational Model

3.13.1 Goals and Principles

Streaming is intended for multimedia applications: video, music, etc. A stream consists of an unterminated series of messages, each message containing a fragment of streamed data.

The AMQP/Fast streaming model separates the compression technology (codec) from the streaming technology. The streaming model is compatible with any codec that allows data to be fragmented.

Streamed messages use the full AMQP/Fast queueing and routing mechanisms with some simplifications:

1. Streamed content is non-persistent though the server MAY hold it on disk if needed.
2. Streamed data sent to a queue cannot be shared between multiple consumers (each would receive partial streams).
3. Streamed data sent to a subscription via a topic cannot be shared between multiple consumers (each would receive partial streams).

A stream producer can use topics to distribute streams to multiple consumers at once. Topics can be used - e.g. - to provide different qualities of the same stream.

A stream consumer can specify the rate at which data is wanted.

3.13.2 Stream Content Properties

These are the properties defined for Stream content:

- ContentType (short string)
- ContentEncoding (short string)
- Priority (octet)
- Destination (short string)
- Timestamp (short string)
- Headers (field table)

3.14 Security

3.14.1 Goals and Principles

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.

3.14.2 Buffer Overflows

All data is length-specified so that applications can allocate memory in advance and avoid deadlocks. Length-specified strings protect against buffer-overflow attacks.

3.14.3 Denial of Service Attacks

AMQP/Fast handles errors by returning a reply code and then closing the channel or connection. This avoids ambiguous states after errors.

It should be assumed that exceptional conditions during connection negotiation 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 clients provoking multiple failures.

4 Reference Section

4.1 The Connection Class

The connection class provides methods for a client to establish a network connection to a server, and for both peers to operate the connection thereafter. The ID of the Connection class is 1.

This is the formal grammar for the class:

```
connection          = open-connection *use-connection close-connection
open-connection     = C:protocol-header
                      S:START C:START-OK
                      *challenge
                      S:TUNE C:TUNE-OK
                      C:OPEN S:OPEN-OK
challenge           = S:SECURE C:SECURE-OK
use-connection      = S:UNKNOWN
                    / C:UNKNOWN
                    / channel
close-connection    = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

This class contains the following server methods:

- Connection.Start-Ok - select security mechanism and locale (ID = 2) (sync reply for Start)
- Connection.Secure-Ok - seurity mechanism response (ID = 4) (sync reply for Secure)
- Connection.Tune-Ok - negotiate connection tuning parameters (ID = 6) (sync reply for Tune)
- Connection.Open - open a path to a virtual host (ID = 7) (sync request)
- Connection.Unknown - signal that an interface test method has failed (ID = 9) (async)
- Connection.Close - request a connection close (ID = 10) (sync request)
- Connection.Close-Ok - confirm a connection close (ID = 11) (sync reply for Close)

This class contains the following client methods:

- Connection.Start - start connection negotiation (ID = 1) (sync request)
- Connection.Secure - seurity mechanism challenge (ID = 3) (sync request)
- Connection.Tune - propose connection tuning parameters (ID = 5) (sync request)
- Connection.Open-Ok - signal that the connection is ready (ID = 8) (sync reply for Open)
- Connection.Unknown - signal that an interface test method has failed (ID = 9) (async)
- Connection.Close - request a connection close (ID = 10) (sync request)
- Connection.Close-Ok - confirm a connection close (ID = 11) (sync reply for Close)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MUST implement this class. Each method may have specific guidelines.
- Any assertion failures in the Connection methods MUST BE treated as connection exceptions - i.e. the peer that detects the error MUST respond with Connection.Close.

4.1.1 The Connection.Start Method

This method starts the connection negotiation process by telling the client the protocol version that the server proposes, along with a list of security mechanisms which the client can use for authentication.

The Connection.Start method has the following specific fields:

- version_major (octet) - The protocol major version that the server agrees to use, which cannot be higher than the client's major version.
- version_minor (octet) - The protocol minor version that the server agrees to use, which cannot be higher than the client's minor version.
- mechanisms (longstr) - A list of the security mechanisms that the server supports, delimited by spaces. Currently AMQP/Fast supports these mechanisms: PLAIN.
- locales (longstr) - A list of the message locales that the server supports, delimited by spaces. The locale defines the language in which the server will send reply texts. All servers MUST support at least the en_US locale.

This is the Connection.Start pseudo-structure:

```
define Connection.Start {
    method_class        octet       #  class ID
    method_id           octet       #  method ID
    method_flags        octet       #  method flags
    method_synctag      long        #  synctag
    version_major       octet       #  negotiated protocol major version
    version_minor       octet       #  negotiated protocol major version
    mechanisms          longstr     #  available security mechanisms
    locales             longstr     #  available message locales
}
```

Guidelines for implementors:

- If the client cannot handle the protocol version suggested by the server it MUST close the socket connection.
- The server MUST provide a protocol version that is lower than or equal to that requested by the client in the protocol header. If the server cannot support the specified protocol it MUST NOT send this method, but MUST close the socket connection.
- The client MUST implement this method.
- This method is a synchronous request that expects one of: Connection.Start-Ok unless there is an exception.
- The "version_major" field MUST be equal to 0.
- The "version_minor" field MUST be equal to 9.
- The "mechanisms" field MUST not be empty.
- The "locales" field MUST not be empty.
- All servers MUST support at least the en_US locale.

4.1.2 The Connection.Start-Ok Method

This method selects a SASL security mechanism. AMQP/Fast uses SASL (RFC2222) to negotiate authentication and encryption.

The Connection.Start-Ok method has the following specific fields:

- mechanism (shortstr) - A single security mechanisms selected by the client, which must be one of those specified by the server. The client SHOULD authenticate using the highest-level security profile it can handle from the list provided by the server.
- locale (shortstr) - A single message local selected by the client, which must be one of those specified by the server.

This is the Connection.Start-Ok pseudo-structure:

```
define Connection.Start-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    mechanism          shortstr    #  selected security mechanism
    locale             shortstr    #  selected message locale
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- The "mechanism" field MUST not be empty.
- The client SHOULD authenticate using the highest-level security profile it can handle from the list provided by the server.
- The "locale" field MUST not be empty.

4.1.3 The Connection.Secure Method

The SASL protocol works by exchanging challenges and responses until both peers have received sufficient information to authenticate each other. This method challenges the client to provide more information.

The Connection.Secure method has the following specific fields:

- challenge (longstr) - Challenge information, a block of opaque binary data passed to the security mechanism.

This is the Connection.Secure pseudo-structure:

```
define Connection.Secure {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    challenge          longstr     #  security challenge data
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous request that expects one of: Connection.Secure-Ok unless there is an exception.
- The "challenge" field MUST not be empty.

4.1.4 The Connection.Secure-Ok Method

This method attempts to authenticate, passing a block of SASL data for the security mechanism at the server side.

The Connection.Secure-Ok method has the following specific fields:

- response (longstr) - A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

This is the Connection.Secure-Ok pseudo-structure:

```
define Connection.Secure-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    response           longstr     #  security response data
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- The "response" field MUST not be empty.

4.1.5 The Connection.Tune Method

This method proposes a set of connection configuration values to the client. The client can accept and/or adjust these.

The Connection.Tune method has the following specific fields:

- frame_max (long) - The largest frame size or fragment size that the server is prepared to accept, in octets. The frame-max must be large enough for any method frame. The frame-max may exceed 64K if the peer is willing to read and write meta frames.
- channel_max (short) - The maximum total number of channels that the server allows per connection. Zero means that the server does not impose a fixed limit, but the number of allowed channels may be limited by available server resources.
- access_max (short) - The maximum total number of access tickets that the server allows per connection. Zero means that the server does not impose a fixed limit, but the number of allowed access tickets may be limited by available server resources.
- consumer_max (short) - The maximum number of consumers that the server allows per channel. Zero means that the server does not impose a fixed limit, but the number of possible consumers may be limited by available server resources.
- heartbeat (short) - The delay, in seconds, of the connection heartbeat that the server wants. Zero means the server does not want a heartbeat.
- txn_limit (short) - The highest number of messages that the server will accept per transaction. Zero means the server does not impose a fixed limit, but the size of transactions may be limited by available server resources.
- jms_support (bit) - Indicates whether the server supports the JMS content domain or not. If the jms-support field is 1, the client MAY use the JMS class methods. If this field is zero, the client MUST NOT use the JMS class methods.
- file_support (bit) - Indicates whether the server supports the file content domain or not. If the file-support field is 1, the client MAY use the File methods. If this field is zero, the client MUST NOT use the File methods.
- stream_support (bit) - Indicates whether the server supports the stream content domain or not. If the stream-support field is 1, the client MAY use the Stream methods. If this field is zero, the client MUST NOT use the Stream methods.

This is the Connection.Tune pseudo-structure:

```
define Connection.Tune {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    frame_max          long        #  maximum frame size
    channel_max        short       #  maximum number of channels
    access_max         short       #  maximum number of access tickets
    consumer_max       short       #  maximum consumers per channel
    heartbeat          short       #  desired heartbeat delay
    txn_limit          short       #  maximum transaction size
    jms_support        bit         #  JMS content is supported?
    file_support       bit         #  file content is supported?
    stream_support     bit         #  stream content is supported?
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous request that expects one of: Connection.Tune-Ok unless there is an exception.
- The ”frame_max” field MUST be greater than or equal to 1024.
- If the jms-support field is 1, the client MAY use the JMS class methods. If this field is zero, the client MUST NOT use the JMS class methods.
- If the file-support field is 1, the client MAY use the File methods. If this field is zero, the client MUST NOT use the File methods.
- If the stream-support field is 1, the client MAY use the Stream methods. If this field is zero, the client MUST NOT use the Stream methods.

4.1.6 The Connection.Tune-Ok Method

This method sends the client's connection tuning parameters to the server. Certain fields are negotiated, others provide capability information.

The Connection.Tune-Ok method has the following specific fields:

- frame_max (long) - The largest frame size or fragment size that the server is prepared to accept, in octets. The frame-max must be large enough for any method frame. The frame-max may exceed 64K if the peer is willing to read and write meta frames. Must not be higher than the value specified by the server.
- channel_max (short) - The maximum total number of channels that the client will use per connection. May not be higher than the value specified by the server. The server MAY ignore the channel-max value or MAY use it for tuning its resource allocation.
- ticket_max (short) - The maximum total number of access tickets that the client will use per connection. May not be higher than the value specified by the server. The server MAY ignore the ticket-max value or MAY use it for tuning its resource allocation.
- heartbeat (short) - The delay, in seconds, of the connection heartbeat that the client wants. Zero means the client does not want a heartbeat.
- jms_support (bit) - Indicates whether the client supports the JMS content domain.
- file_support (bit) - Indicates whether the client supports the file content domain.
- stream_support (bit) - Indicates whether the client supports the stream content domain.

- prefetch_max (long) - This value governs the total amount of prefetch data that the client is willing to accept per connection. Prefetching is the technique of sending messages to a client in advance, which reduces the latency of message processing. Normally each comnsumer specifies its own prefetch window. The prefetch-max field specifies a global limit at the connection level. If it is zero, the server does not impose any connection-level limit. The server MUST NOT send a client more data in advance than this value allows. If sending a specific message in advance would exhaust the connection prefetch window, it MUST NOT send the message. Setting this field to a very low non-zero value - e.g. 1 - effectively disables all prefetching.

This is the Connection.Tune-Ok pseudo-structure:

```
define Connection.Tune-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    frame_max          long        #  maximum frame size
    channel_max        short       #  maximum number of channels
    ticket_max         short       #  maximum number of access tickets
    heartbeat          short       #  desired heartbeat delay
    jms_support        bit         #  JMS content is supported?
    file_support       bit         #  file content is supported?
    stream_support     bit         #  stream content is supported?
    prefetch_max       long        #  maximum prefetch size for connection
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- The ”frame_max” field MUST be greater than or equal to 1024.
- The ”frame_max” field MUST be less than or equal to the value of ’frame max’ in the last received Connection.Tune method.
- The ”channel_max” field MUST be greater than 0.
- The ”channel_max” field MUST be less than or equal to the value of ’channel max’ in the last received Connection.Tune method.
- The server MAY ignore the channel-max value or MAY use it for tuning its resource allocation.
- The ”ticket_max” field MUST be greater than 0.
- The ”ticket_max” field MUST be less than or equal to the value of ’access max’ in the last received Connection.Tune method.
- The server MAY ignore the ticket-max value or MAY use it for tuning its resource allocation.
- The server MUST NOT send a client more data in advance than this value allows. If sending a specific message in advance would exhaust the connection prefetch window, it MUST NOT send the message. Setting this field to a very low non-zero value - e.g. 1 - effectively disables all prefetching.

4.1.7 The Connection.Open Method

This method opens a path to a virtual host on the server. The virtual host is a collection of queues, and acts to separate multiple application domains on the server.

The Connection.Open method has the following specific fields:

- virtual_path (shortstr) - Must start with a slash "/" and continue with path names separated by slashes. A path name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_+!=:]. The virtual access path of the virtual host to work with. The virtual path must be known to the server, either as a configured value or as a built-in value.
- client_id (shortstr) - The client identifier, used to identify persistent resources belonging to the client. This is a string that uniquely defines the client. The client MUST supply an ID that it used previously if it wants to continue using previously-allocated resources. If the client-id is empty, the server MUST allocate a new ID that uniquely and persistently identifies the client to the server instance. The server MUST restrict a specific client identifier to being active in at most one connection at a time. The server SHOULD detect when a client disconnects and release all temporary resources owned by that client.

This is the Connection.Open pseudo-structure:

```
define Connection.Open {
    method_class       octet      #  class ID
    method_id          octet      #  method ID
    method_flags       octet      #  method flags
    method_synctag     long       #  synctag
    virtual_path       shortstr   #  path value virtual server path
    client_id          shortstr   #  client identifier
}
```

Guidelines for implementors:

- The server MUST support the default virtual host, "/".
- The server SHOULD verify that the client has permission to access the specified virtual host, using the authenticated client identity.
- The client MUST open a path to a virtual host before doing any work on the connection.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Connection.Open-Ok unless there is an exception.
- The client MUST supply an ID that it used previously if it wants to continue using previously-allocated resources.
- If the client-id is empty, the server MUST allocate a new ID that uniquely and persistently identifies the client to the server instance.
- The server MUST restrict a specific client identifier to being active in at most one connection at a time.
- The server SHOULD detect when a client disconnects and release all temporary resources owned by that client.

4.1.8 The Connection.Open-Ok Method

This method signals to the client that the connection is ready for use.

The Connection.Open-Ok method has the following specific fields:

- client_id (shortstr) - Confirms or provides the client id. If the client provided an id when sending Connection.Open, this field confirms the ID. If the client did not provide an ID, the server generates one and provides it in this field.

This is the Connection.Open-Ok pseudo-structure:

```
define Connection.Open-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    client_id          shortstr    #  assigned client identifier
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- The ”client_id” field MUST not be empty.

4.1.9 The Connection.Unknown Method

This method signals that an interface test method has failed. This may happen after a method is sent with the UNREAL flag set.

The Connection.Unknown method has the following specific fields:

- class (octet) - The class id of the interface test method that failed.
- method (octet) - The method id of the interface test method that failed.
- synchtag (short) - The synchtag the interface test method that failed.
- reply_code (short) - The reply code. The AMQ reply codes are defined in AMQ RFC 011.
- reply_text (shortstr) - The localised reply text. This text can be logged as an aid to resolving issues.

This is the Connection.Unknown pseudo-structure:

```
define Connection.Unknown {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    class              octet       #  failing method class
    method             octet       #  failing method ID
    synchtag           short       #  failing method synchtag
    reply_code         short       #  reply code from server reply code
    reply_text         shortstr    #  localised reply text localised reply text
}
```

Guidelines for implementors:

- A peer that uses the UNREAL flag MUST implement this method.
- The client SHOULD implement this method.
- The server SHOULD implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- The ”class” field MUST NOT be equal to 0.
- The ”method” field MUST NOT be equal to 0.

4.1.10 The Connection.Close Method

This method indicates that the sender wants to close the connection. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class, method id, and synchtag of the method which caused the exception.

The Connection.Close method has the following specific fields:

- reply_code (short) - The reply code. The AMQ reply codes are defined in AMQ RFC 011.
- reply_text (shortstr) - The localised reply text. This text can be logged as an aid to resolving issues.
- class (octet) - When the close is provoked by a method exception, this is the class of the method.
- method (octet) - When the close is provoked by a method exception, this is the ID of the method.
- synchtag (short) - When the close is provoked by a method exception, this is the synchtag of the method.

This is the Connection.Close pseudo-structure:

```
define Connection.Close {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    reply_code         short       #  reply code from server reply code
    reply_text         shortstr    #  localised reply text localised reply text
    class              octet       #  failing method class
    method             octet       #  failing method ID
    synchtag           short       #  failing method synchtag
}
```

Guidelines for implementors:

- After sending this method any received method except Connection.Close-OK MUST be discarded.
- The peer sending this method MAY use a counter or timeout to detect failure of the other peer to respond correctly with Connection.Close-OK.
- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Connection.Close-Ok unless there is an exception.

4.1.11 The Connection.Close-Ok Method

This method confirms a Connection.Close method and tells the recipient that it is safe to release resources for the connection and close the socket. This method has no fields apart from the standard method header.

This is the Connection.Close-Ok pseudo-structure:

```
define Connection.Close-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- A peer that detects a socket closure without having received a Connection.Close-Ok handshake method SHOULD log the error.

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.2 The Channel Class

The channel class provides methods for a client to establish a virtual connection - a channel - to a server and for both peers to operate the virtual connection thereafter. The ID of the Channel class is 2.

This is the formal grammar for the class:

```
channel             = open-channel *use-channel close-channel
open-channel        = C:OPEN S:OPEN-OK
use-channel         = C:FLOW
                    / S:FLOW
                    / S:SYNCH
                    / access
                    / proxy
                    / queue
                    / jms
                    / file
                    / stream
                    / tx
                    / dtx
                    / test
close-channel       = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

This class contains the following server methods:

- Channel.Open - open a channel for use (ID = 1) (sync request)
- Channel.Flow - enable/disable flow from peer (ID = 3) (async)
- Channel.Close - request a channel close (ID = 4) (sync request)
- Channel.Close-Ok - confirm a channel close (ID = 5) (sync reply for Close)

This class contains the following client methods:

- Channel.Open-Ok - signal that the channel is ready (ID = 2) (sync reply for Open)
- Channel.Flow - enable/disable flow from peer (ID = 3) (async)
- Channel.Close - request a channel close (ID = 4) (sync request)
- Channel.Close-Ok - confirm a channel close (ID = 5) (sync reply for Close)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MUST implement this class. Each method may have specific guidelines.
- Any assertion failures in the Channel methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.2.1 The Channel.Open Method

Client asks server to open a new channel.

The Channel.Open method has the following specific fields:

- prefetch_max (long) - This value governs the total amount of prefetch data that the client is willing to accept per channel. Prefetching is the technique of sending messages to a client in advance, which reduces the latency of message processing. Normally each comnsumer specifies its own prefetch window. The prefetch-max field specifies an overall limit at the channel level. If it is zero, the server does not impose any channel-level limit. The server MUST NOT send a client more data in advance than this value allows. If sending a specific message in advance would exhaust the channel prefetch window, it MUST NOT send the message. Setting this field to a very low non-zero value - e.g. 1 - effectively disables all prefetching on the channel.
- out_of_band (shortstr) - Configures out-of-band transfers on this channel. The syntax and meaning of this field will be formally defined at a later date.
- tx_mode (octet) - Configures the transaction mode for the channel. AMQP/Fast supports three transactional models: automatic, standard, and distributed (e.g. XA).

This is the Channel.Open pseudo-structure:

```
define Channel.Open {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    prefetch_max       long        #  maximum prefetch size for channel
    out_of_band        shortstr    #  out-of-band settings for channel
    tx_mode            octet       #  transaction mode for channel
}
```

Guidelines for implementors:

- This method is not allowed when the channel is already open.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Channel.Open-Ok unless there is an exception.
- The server MUST NOT send a client more data in advance than this value allows. If sending a specific message in advance would exhaust the channel prefetch window, it MUST NOT send the message. Setting this field to a very low non-zero value - e.g. 1 - effectively disables all prefetching on the channel.
- The ”out_of_band” field MUST be empty.
- The ”tx_mode” field MUST be one of: 1=automatic (value.), 2=standard (value.), 3=distributed (value.).

4.2.2 The Channel.Open-Ok Method

This method signals to the client that the channel is ready for use. This method has no fields apart from the standard method header.

This is the Channel.Open-Ok pseudo-structure:

```
define Channel.Open-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.

- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.2.3 The Channel.Flow Method

This method asks the peer to pause or restart the flow of content data. This is a simple flow-control mechanism that a peer can use to avoid oveflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control.

The Channel.Flow method has the following specific fields:

- flow_pause (bit) - If 1, the peer stops sending content frames. If 0, the peer restarts sending content frames.

This is the Channel.Flow pseudo-structure:

```
define Channel.Flow {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    flow_pause         bit         #  start/stop content frames
}
```

Guidelines for implementors:

- When sending content data in multiple frames, a peer SHOULD monitor the channel for incoming methods and respond to a Channel.Flow as rapidly as possible.
- A peer MAY use the Channel.Flow method to throttle incoming content data for internal reasons, for example, when proxying data over a slower connection.
- The peer that requests a Channel.Flow method MAY disconnect and/or ban a peer that does not respect the request.
- The server MUST implement this method.
- The client MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).

4.2.4 The Channel.Close Method

This method indicates that the sender wants to close the channel. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class, method id, and synchtag of the method which caused the exception.

The Channel.Close method has the following specific fields:

- reply_code (short) - The reply code. The AMQ reply codes are defined in AMQ RFC 011.
- reply_text (shortstr) - The localised reply text. This text can be logged as an aid to resolving issues.
- class (octet) - When the close is provoked by a method exception, this is the class of the method.
- method (octet) - When the close is provoked by a method exception, this is the ID of the method.
- synchtag (short) - When the close is provoked by a method exception, this is the synchtag of the method.

This is the Channel.Close pseudo-structure:

```
define Channel.Close {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    reply_code         short       #  reply code from server reply code
    reply_text         shortstr    #  localised reply text localised reply text
    class              octet       #  failing method class
    method             octet       #  failing method ID
    synchtag           short       #  failing method synchtag
}
```

Guidelines for implementors:

- After sending this method any received method except Channel.Close-OK MUST be discarded.
- The peer sending this method MAY use a counter or timeout to detect failure of the other peer to respond correctly with Channel.Close-OK..
- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Channel.Close-Ok unless there is an exception.

4.2.5 The Channel.Close-Ok Method

This method confirms a Channel.Close method and tells the recipient that it is safe to release resources for the channel and close the socket. This method has no fields apart from the standard method header.

This is the Channel.Close-Ok pseudo-structure:

```
define Channel.Close-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- A peer that detects a socket closure without having received a Channel.Close-Ok handshake method SHOULD log the error.
- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.3 The Access Class

AMQP/Fast controls access to server resources using access tickets. A client must explicitly request access tickets before doing work. An access ticket grants a client the right to use a specific set of resources - called a "realm" - in specific ways. The ID of the Access class is 3.

This is the formal grammar for the class:

```
access              = C:GRANT S:GRANT-OK
```

This class contains the following server methods:

- Access.Grant - request an access ticket (ID = 1) (sync request)

This class contains the following client methods:

- Access.Grant-Ok - grant access to server resources (ID = 2) (sync reply for Grant)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MUST implement this class. Each method may have specific guidelines.
- Any assertion failures in the Access methods MUST BE treated as connection exceptions - i.e. the peer that detects the error MUST respond with Connection.Close.

4.3.1 The Access.Grant Method

This method requests an access ticket for an access realm. The server responds by granting the access ticket. If the client does not have access rights to the requested realm this causes a connection exception. Access tickets may be shared across channels within a connection and expire with the connection.

The Access.Grant method has the following specific fields:

- realm (shortstr) - Must start with a slash "/" and continue with path names separated by slashes. A path name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_+!=:].
- exclusive (bit) - Request exclusive access to the realm. If the server cannot grant this - because there are other active tickets for the realm - it raises a channel exception. The server MUST grant clients exclusive access to a realm if they ask for it.
- passive (bit) - Request message passive access to the specified access realm. Passive access lets a client get information about resources in the realm but not to make any changes to them.
- active (bit) - Request message active access to the specified access realm. Acvtive access lets a client get create and delete resources in the realm.
- write (bit) - Request write access to the specified access realm. Write access lets a client publish messages to all proxies in the realm.
- read (bit) - Request read access to the specified access realm. Read access lets a client consume messages from queues in the realm.

This is the Access.Grant pseudo-structure:

```
define Access.Grant {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    realm              shortstr    #  path value realm to work with
    exclusive          bit         #  request exclusive access
    passive            bit         #  request passive access
    active             bit         #  request active access
    write              bit         #  request write access
    read               bit         #  request read access
}
```

Guidelines for implementors:

- The realm name MUST start with either "/data" (for application resources) or "/admin" (for server administration resources).
- The server MUST implement the /data realm and MAY implement the /admin realm. The mapping of resources to realms is not defined in the protocol - this is a server-side configuration issue.

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Access.Grant-Ok unless there is an exception.
- The server MUST grant clients exclusive access to a realm if they ask for it.

4.3.2 The Access.Grant-Ok Method

This method provides the client with an access ticket. The access ticket is valid within the current channel and for the lifespan of the channel.

The Access.Grant-Ok method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection.

This is the Access.Grant-Ok pseudo-structure:

```
define Access.Grant-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
}
```

Guidelines for implementors:

- The client MUST NOT use access tickets except within the same channel as originally granted.
- The server MUST isolate access tickets per channel and treat an attempt by a client to mix these as a connection exception.
- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.4 The Proxy Class

Proxies match and distribute messages across queues. Proxies can be configured in the server or created at runtime. The ID of the Proxy class is 4.

This is the formal grammar for the class:

```
proxy               = C:ASSERT   S:ASSERT-OK
                    / C:DELETE   S:DELETE-OK
```

This class contains the following server methods:

- Proxy.Assert - create proxy if needed (ID = 1) (sync request)
- Proxy.Delete - delete a proxy (ID = 3) (sync request)

This class contains the following client methods:

- Proxy.Assert-Ok - confirms a proxy definition (ID = 2) (sync reply for Assert)
- Proxy.Delete-Ok - confirm deletion of a proxy (ID = 4) (sync reply for Delete)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MUST implement this class. Each method may have specific guidelines.

- Any assertion failures in the Proxy methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.4.1 The Proxy.Assert Method

This method creates or checks a proxy. When creating a new proxy the client can specify various properties that control the durability of the proxy and the level of sharing for the proxy.

The Proxy.Assert method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. When a client defines a new proxy, this belongs to the access realm of the ticket used. All further work done with that proxy must be done with an access ticket for the same realm. The client MUST provide a valid access ticket giving "active" access to the realm in which the proxy exists or will be created, or "passive" access if the if-exists flag is set.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive.
- class (shortstr) - Specifies the name of a proxy class, which is a server defined class that defines the proxy functionality. Proxy classes are standardised but can be extended on a per-implementation basis.
- if_exists (bit) - If set, the server will report the status of a proxy if it exists and raise a channel assertion if not. This flag lets clients discover the status of non-existent proxies without creating them.
- durable (bit) - If set when creating a new proxy, the proxy will be marked as durable. Durable proxies remain active when a server restarts. Non-durable proxies (transient proxies) are purged if/when a server restarts. The server MUST support both durable and transient proxies. The server MUST ignore the durable field if the proxy already exists.
- auto_delete (bit) - If set, the proxy is deleted when all queues have finished using it. The server MUST implement the auto-delete function in this manner: it counts the number of queue registrations for the proxy and when the last queue registration is cancelled, it MUST delete the proxy. The server MUST ignore the auto-delete field if the proxy already exists.

This is the Proxy.Assert pseudo-structure:

```
define Proxy.Assert {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    proxy              shortstr    #  proxy name
    class              shortstr    #  proxy class
    if_exists          bit         #  do not create proxy
    durable            bit         #  request a durable proxy
    auto_delete        bit         #  auto-delete proxy when unused
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Proxy.Assert-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "active" access to the realm in which the proxy exists or will be created, or "passive" access if the if-exists flag is set.
- The "proxy" field MUST not be empty.
- The "class" field MUST not be empty.

- The server MUST support both durable and transient proxies.
- The server MUST ignore the durable field if the proxy already exists.
- The server MUST implement the auto-delete function in this manner: it counts the number of queue registrations for the proxy and when the last queue registration is cancelled, it MUST delete the proxy.
- The server MUST ignore the auto-delete field if the proxy already exists.

4.4.2 The Proxy.Assert-Ok Method

This method confirms a Assert method and confirms the name of the proxy, essential for automatically-named proxies. This method has no fields apart from the standard method header.

This is the Proxy.Assert-Ok pseudo-structure:

```
define Proxy.Assert-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.4.3 The Proxy.Delete Method

This method deletes a proxy. When a proxy is deleted all queue registrations on the proxy are cancelled.

The Proxy.Delete method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "active" access rights to the proxy's access realm.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. The proxy must exist. Attempting to delete a non-existing proxy causes a channel exception.
- if_unused (bit) - If set, the server will only delete the proy if it has no queue registrations. If the proxy has queue registrations the server does does not delete it but raises a channel exception instead.

This is the Proxy.Delete pseudo-structure:

```
define Proxy.Delete {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    proxy              shortstr    #  proxy name
    if_unused          bit         #  delete only if unused
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Proxy.Delete-Ok unless there is an exception.

- The client MUST provide a valid access ticket giving ”active” access rights to the proxy’s access realm.
- The ”proxy” field MUST not be empty.
- The proxy must exist. Attempting to delete a non-existing proxy causes a channel exception.

4.4.4 The Proxy.Delete-Ok Method

This method confirms the deletion of a proxy. This method has no fields apart from the standard method header.

This is the Proxy.Delete-Ok pseudo-structure:

```
define Proxy.Delete-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.5 The Queue Class

Queues store and forward messages. Queues can be configured in the server or created at runtime. Queues must be attached to at least one proxy in order to receive messages from publishers. The ID of the Queue class is 5.

This is the formal grammar for the class:

```
queue               = C:ASSERT   S:ASSERT-OK
                    / C:REGISTER S:REGISTER-OK
                    / C:CANCEL   S:CANCEL-OK
                    / C:PURGE    S:PURGE-OK
                    / C:DELETE   S:DELETE-OK
```

This class contains the following server methods:

- Queue.Assert - create queue if needed (ID = 1) (sync request)
- Queue.Register - register queue with a proxy (ID = 3) (sync request)
- Queue.Purge - purge a queue (ID = 5) (sync request)
- Queue.Delete - delete a queue (ID = 7) (sync request)

This class contains the following client methods:

- Queue.Assert-Ok - confirms a queue definition (ID = 2) (sync reply for Assert)
- Queue.Register-Ok - confirm registration successful (ID = 4) (sync reply for Register)
- Queue.Purge-Ok - confirms a queue purge (ID = 6) (sync reply for Purge)
- Queue.Delete-Ok - confirm deletion of a queue (ID = 8) (sync reply for Delete)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MUST implement this class. Each method may have specific guidelines.

- Any assertion failures in the Queue methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.5.1 The Queue.Assert Method

This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

The Queue.Assert method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. When a client defines a new queue, this belongs to the access realm of the ticket used. All further work done with that queue must be done with an access ticket for the same realm. The client MUST provide a valid access ticket giving "active" access to the realm in which the queue exists or will be created, or "passive" access if the if-exists flag is set.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. The queue name MAY be empty, in which case the server MUST create a new queue with a unique generated name and return this to the client in the Assert-Ok method. Clients asking the server to provide a queue name SHOULD use a specific namespace for such queues to avoid any possible conflict with names that might be used elsewhere by clients.
- template (shortstr) - Specifies the name of a queue template, which is a server configured object that provides queue configuration options. Template semantics and names are not standardised. If the template is empty the server SHOULD use a suitable default. The server MUST ignore the template field if the queue already exists.
- if_exists (bit) - If set, the server will report the status of a queue if it exists and raise a channel assertion if not. This flag lets clients discover the status of non-existent queues without creating them.
- durable (bit) - If set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. The server MUST support both durable and transient queues. The server MUST ignore the durable field if the queue already exists.
- private (bit) - If set when creating a new queue, the queue will be private and owned by the current client. This will fail if the queue already exists and is owned by another client. Private queues cannot be consumed from by clients except the owner. The server MUST support both private and shared queues. The server MUST use the client identifier supplied at connection open time to identify the owner of a private queue. The client identifier is persistent even if the client disconnects and reconnects. The server MUST ignore the private field if the queue already exists.
- auto_delete (bit) - If set, the queue is deleted when all clients have finished using it. The server MUST implement the auto-delete function in this manner: it counts the number of queue consumers and when the last consumer is cancelled, it MUST delete the queue and dead-letter any messages it holds. The server MUST ignore the auto-delete field if the queue already exists.

This is the Queue.Assert pseudo-structure:

```
define Queue.Assert {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    template           shortstr    #  queue template
    if_exists          bit         #  do not create queue
    durable            bit         #  request a durable queue
    private            bit         #  request a private queue
    auto_delete        bit         #  auto-delete queue when unused
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Queue.Assert-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "active" access to the realm in which the queue exists or will be created, or "passive" access if the if-exists flag is set.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The queue name MAY be empty, in which case the server MUST create a new queue with a unique generated name and return this to the client in the Assert-Ok method.
- Clients asking the server to provide a queue name SHOULD use a specific namespace for such queues to avoid any possible conflict with names that might be used elsewhere by clients.
- If the template is empty the server SHOULD use a suitable default.
- The server MUST ignore the template field if the queue already exists.
- The server MUST support both durable and transient queues.
- The server MUST ignore the durable field if the queue already exists.
- The server MUST support both private and shared queues.
- The server MUST use the client identifier supplied at connection open time to identify the owner of a private queue. The client identifier is persistent even if the client disconnects and reconnects.
- The server MUST ignore the private field if the queue already exists.
- The server MUST implement the auto-delete function in this manner: it counts the number of queue consumers and when the last consumer is cancelled, it MUST delete the queue and dead-letter any messages it holds.
- The server MUST ignore the auto-delete field if the queue already exists.

4.5.2 The Queue.Assert-Ok Method

This method confirms a Assert method and confirms the name of the queue, essential for automatically-named queues.

The Queue.Assert-Ok method has the following specific fields:

- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Reports the name of the queue. If the server generated a queue name, this field contains that name.

- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Reports the name of the last-registered proxy for the queue. In the case of newly-created queues this will be empty. The server MUST NOT register newly-created queues with proxies implicitly or automatically. The decision of what proxies to use is taken by the client application responsible for administrating the server. However the server SHOULD store durable registrations so that on a server restart these can be recreated.
- message_count (long) - Reports the number of messages in the queue, which will be zero for newly-created queues.
- consumer_count (long) - Reports the number of active consumers for the queue. Note that consumers can suspend activity (Channel.Flow) in which case they do not appear in this count.

This is the Queue.Assert-Ok pseudo-structure:

```
define Queue.Assert-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    queue              shortstr    #  queue name name of queue
    proxy              shortstr    #  proxy name queue proxy, if any
    message_count      long        #  number of messages in queue queue
    consumer_count     long        #  number of consumers
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- The ”queue” field MUST not be empty.
- The server MUST NOT register newly-created queues with proxies implicitly or automatically. The decision of what proxies to use is taken by the client application responsible for administrating the server. However the server SHOULD store durable registrations so that on a server restart these can be recreated.

4.5.3 The Queue.Register Method

This method registers a queue with a proxy. Until a queue is registered it will not receive any messages. The two main types of queue are store-and-forward queues (registered with a dest-name proxy) and subscription queues (registered with a dest-wild proxy).

The Queue.Register method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving ”active” access rights to the queue’s access realm.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.

- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. The queue must exist. Attempting to query a non-existing queue causes a channel exception.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. The name of the proxy to register with. If the proxy does not exist the server will raise a channel exception.
- arguments (table) - A set of arguments for the proxy. The syntax and semantics of these arguments depends on the proxy class.

This is the Queue.Register pseudo-structure:

```
define Queue.Register {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    proxy              shortstr    #  proxy name proxy to register with
    arguments          table       #  arguments for registration
}
```

Guidelines for implementors:

- The server MUST NOT allow the registration of a durable queue with a transient proxy. If the client attempts this the server MUST raise a channel exception.
- Registrations for durable queues are automatically durable and the server SHOULD restore such registrations after a server restart.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Queue.Register-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "active" access rights to the queue's access realm.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The queue must exist. Attempting to query a non-existing queue causes a channel exception.

4.5.4 The Queue.Register-Ok Method

This method confirms that a registration was successful. This method has no fields apart from the standard method header.

This is the Queue.Register-Ok pseudo-structure:

```
define Queue.Register-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.5.5 The Queue.Purge Method

This method removes all messages from a queue. It does not cancel consumers. Purged messages are deleted without any formal "undo" mechanism.

The Queue.Purge method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The access ticket must be for the access realm that holds the queue. The client MUST provide a valid access ticket giving "read" access rights to the queue's access realm. Note that purging a queue is equivalent to reading all messages and discarding them.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. The queue must exist. Attempting to purge a non-existing queue causes a channel exception.

This is the Queue.Purge pseudo-structure:

```
define Queue.Purge {
    method_class        octet       #  class ID
    method_id           octet       #  method ID
    method_flags        octet       #  method flags
    method_synctag      long        #  synctag
    ticket              short       #  access ticket granted by server
    namespace           shortstr    #  queue namespace
    queue               shortstr    #  queue name
}
```

Guidelines for implementors:

- On transacted channels the server MUST not purge messages that have already been sent to a client but not yet acknowledged.
- The server MAY implement a purge queue or log that allows system administrators to recover accidentally-purged messages. The server SHOULD NOT keep purged messages in the same storage spaces as the live messages since the volumes of purged messages may get very large.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Queue.Purge-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "read" access rights to the queue's access realm. Note that purging a queue is equivalent to reading all messages and discarding them.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The queue must exist. Attempting to purge a non-existing queue causes a channel exception.

4.5.6 The Queue.Purge-Ok Method

This method confirms the purge of a queue.

The Queue.Purge-Ok method has the following specific fields:

- message_count (long) - Reports the number of messages purged.

This is the Queue.Purge-Ok pseudo-structure:

```
define Queue.Purge-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    message_count      long        #  number of messages purged
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.5.7 The Queue.Delete Method

This method deletes a queue. When a queue is deleted any pending messages are sent to a dead-letter queue if this is defined in the server configuration, and all consumers on the queue are cancelwled.

The Queue.Delete method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "active" access rights to the queue's access realm.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. The queue must exist. Attempting to delete a non-existing queue causes a channel exception.
- if_unused (bit) - If set, the server will only delete the queue if it has no consumers. If the queue has consumers the server does does not delete it but raises a channel exception instead.
- if_empty (bit) - If set, the server will only delete the queue if it has no messages. If the queue is not empty the server raises a channel exception.

This is the Queue.Delete pseudo-structure:

```
define Queue.Delete {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    if_unused          bit         #  delete only if unused
    if_empty           bit         #  delete only if empty
}
```

Guidelines for implementors:

- The server SHOULD use a dead-letter queue to hold messages that were pending on a deleted queue, and MAY provide facilities for a system administrator to move these messages back to an active queue.
- The server MUST implement this method.

- This method is a synchronous request that expects one of: Queue.Delete-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "active" access rights to the queue's access realm.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The queue must exist. Attempting to delete a non-existing queue causes a channel exception.

4.5.8 The Queue.Delete-Ok Method

This method confirms the deletion of a queue.

The Queue.Delete-Ok method has the following specific fields:

- message_count (long) - Reports the number of messages purged.

This is the Queue.Delete-Ok pseudo-structure:

```
define Queue.Delete-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    message_count      long        #  number of messages purged
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.6 The Jms Class

The JMS class provides methods that support the standard JMS API. JMS messages have a specific set of properties that are required for interoperability with JMS providers and consumers. JMS messages and acknowledgements are subject to channel transactions. The ID of the Jms class is 6.

This is the formal grammar for the class:

```
jms                  = C:CONSUME S:CONSUME-OK
                     / C:CANCEL S:CANCEL-OK
                     / C:PUBLISH content
                     / S:DELIVER content
                     / C:BROWSE ( S:BROWSE-OK content / S:BROWSE_EMPTY )
                     / C:ACK
                     / C:REJECT
```

This class contains the following server methods:

- Jms.Consume - start a queue consumer (ID = 1) (sync request)
- Jms.Cancel - end a queue consumer (ID = 3) (sync request)
- Jms.Publish - publish a message (ID = 5) (async, carries content)
- Jms.Browse - direct access to a queue (ID = 7) (sync request)
- Jms.Ack - acknowledge one or more messages (ID = 10) (async)
- Jms.Reject - reject an incoming message (ID = 11) (async)

This class contains the following client methods:

- Jms.Consume-Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- Jms.Cancel-Ok - confirm a cancelled consumer (ID = 4) (sync reply for Cancel)
- Jms.Deliver - notify the client of a consumer message (ID = 6) (async, carries content)
- Jms.Browse-Ok - provide client with a browsed message (ID = 8) (sync reply for Browse, carries content)
- Jms.Browse Empty - indicate no messages available (ID = 9) (sync reply for Browse)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client MAY implement this class. Each method may have specific guidelines.
- Any assertion failures in the Jms methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.6.1 The Jms.Consume Method

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were created on, or until the client cancels them.

The Jms.Consume method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue to consume from.
- prefetch_size (short) - The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. May be set to zero, meaning "no specific limit". Note that other prefetch limits may still apply.
- prefetch_count (short) - Specifies a prefetch window in terms of whole messages. This is compatible with some JMS API implementations. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.
- no_local (bit) - If this field is set the server will not send messages to the client that published them.

- auto_ack (bit) - If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.
- exclusive (bit) - Request exclusive consumer access. If the server cannot grant this - because there are other consumers active - it raises a channel exception. The server MUST grant clients exclusive access to a queue if they ask for it.

This is the Jms.Consume pseudo-structure:

```
define Jms.Consume {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    prefetch_size      short       #  prefetch window in octets
    prefetch_count     short       #  prefetch window in messages
    no_local           bit         #  do not receive own messages
    auto_ack           bit         #  no acknowledgement needed
    exclusive          bit         #  request exclusive access
}
```

Guidelines for implementors:

- The server MAY restrict the number of consumers per channel to an arbitrary value, which MUST be at least 8, and MUST be specified in the Connection.Tune method.
- The client MUST be able to work with the server-defined limits with respect to the maximum number of consumers per channel.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Jms.Consume-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.
- The server MUST grant clients exclusive access to a queue if they ask for it.

4.6.2 The Jms.Consume-Ok Method

This method provides the client with a consumer tag which it MUST use in methods that work with the consumer.

The Jms.Consume-Ok method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the Jms.Consume-Ok pseudo-structure:

```
define Jms.Consume-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.6.3 The Jms.Cancel Method

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

The Jms.Cancel method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the Jms.Cancel pseudo-structure:

```
define Jms.Cancel {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Jms.Cancel-Ok unless there is an exception.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.6.4 The Jms.Cancel-Ok Method

This method confirms that the cancellation was completed. This method has no fields apart from the standard method header.

This is the Jms.Cancel-Ok pseudo-structure:

```
define Jms.Cancel-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.6.5 The Jms.Publish Method

This method publishes a message to a specific proxy. The message will be forwarded to all registered queues and distributed to any active consumers when the transaction is committed.

The Jms.Publish method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy to publish to. If the proxy does not exist the server will raise a channel exception.
- immediate (bit) - Asserts that the message is delivered to one or more consumers immediately and causes a channel exception if this is not the case.

This is the Jms.Publish pseudo-structure:

```
define Jms.Publish {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    proxy              shortstr    #  proxy name
    immediate          bit         #  assert immediate delivery
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- The "proxy" field MUST not be empty.

4.6.6 The Jms.Deliver Method

This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

The Jms.Deliver method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- redelivered (bit) - This indicates that the message has been previously delivered to this or another client. The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy that the message was originally published to.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue that the message came from. Note that a single channel can start many consumers on different queues.

This is the Jms.Deliver pseudo-structure:

```
define Jms.Deliver {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    redelivered        bit         #  signal a redelivered message
    proxy              shortstr    #  proxy name
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
}
```

Guidelines for implementors:

- The server SHOULD track the number of times a message has been delivered to clients and when a message is redelivered a certain number of times - e.g. 5 times - without being acknowledged, the server SHOULD consider the message to be unprocessable (possibly causing client applications to abort), and move the message to a dead letter queue.
- The client MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.

- The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.
- The ”proxy” field MUST not be empty.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The ”queue” field MUST not be empty.

4.6.7 The Jms.Browse Method

This method provides a direct access to the messages in a queue using a synchronous dialogue that is designed for specific types of application where functionality is more important than performance.

The Jms.Browse method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving ”read” access rights to the realm for the queue.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue to browse from.
- no_local (bit) - If this field is set the server will not send messages to the client that published them.
- auto_ack (bit) - If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.

This is the Jms.Browse pseudo-structure:

```
define Jms.Browse {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    no_local           bit         #  do not receive own messages
    auto_ack           bit         #  no acknowledgement needed
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Jms.Browse-Ok Jms.Browse Empty unless there is an exception.
- The client MUST provide a valid access ticket giving ”read” access rights to the realm for the queue.

- The queue namespace MAY be empty - the empty namespace acts just like any other name space.

4.6.8 The Jms.Browse-Ok Method

This method delivers a message to the client following a browse method. A browsed message will need to be acknowkedged, unless the auto-ack option was set to one.

The Jms.Browse-Ok method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- redelivered (bit) - This indicates that the message has been previously delivered to this or another client. The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.
- message_count (long) - This field reports the number of messages pending on the queue, excluding the message being delivered. Note that this figure is indicative, not reliable, and can change arbitrarily as messages are added to the queue and removed by other clients.

This is the Jms.Browse-Ok pseudo-structure:

```
define Jms.Browse-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    redelivered        bit         #  signal a redelivered message
    message_count      long        #  number of messages pending
}
```

Guidelines for implementors:

- The client MAY implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.

4.6.9 The Jms.Browse Empty Method

This method tells the client that the queue has no messages available for the client. This method has no fields apart from the standard method header.

This is the Jms.Browse Empty pseudo-structure:

```
define Jms.Browse Empty {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MAY implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.6.10 The Jms.Ack Method

This method acknowledges one or more messages delivered via the Deliver or Browse-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

The Jms.Ack method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- multiple (bit) - If set to 1, the delivery tag is treated as "up to and including", so that the client can acknowledge multiple messages with a single method. If set to zero, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, tells the server to acknowledge all outstanding mesages. The server MUST validate that a non-zero delivery-tag refers to an delivered message, and raise a channel exception if this is not the case.

This is the Jms.Ack pseudo-structure:

```
define Jms.Ack {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    multiple           bit         #  acknowledge multiple messages
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The server MUST validate that a non-zero delivery-tag refers to an delivered message, and raise a channel exception if this is not the case.

4.6.11 The Jms.Reject Method

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

The Jms.Reject method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- requeue (bit) - If this field is zero, the message will be discarded. If this bit is 1, the server will attempt to requeue the message. The server MUST NOT deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

This is the Jms.Reject pseudo-structure:

```
define Jms.Reject {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    requeue            bit         #  requeue the message
}
```

Guidelines for implementors:

- The server SHOULD be capable of accepting and process the Reject method while sending message content with a Deliver or Browse-Ok method. I.e. the server should read and process incoming methods while sending output frames.
- The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.
- A client MUST NOT use this method as a means of selecting messages to process. A rejected message MAY be discarded or dead-lettered, not necessarily passed to another client.
- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The server MUST NOT deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

4.7 The File Class

The file class provides methods that support reliable file transfer. File messages have a specific set of properties that are required for interoperability with file transfer applications. File messages and acknowledgements are subject to channel transactions. Note that the file class does not provide message browsing methods; these are not compatible with the staging model. Applications that need browsable file transfer should use JMS content and the JMS class. The ID of the File class is 7.

This is the formal grammar for the class:

```
file                = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:OPEN S:OPEN-OK C:STAGE content
                    / S:OPEN C:OPEN-OK S:STAGE
                    / C:PUBLISH
                    / S:DELIVER
                    / C:ACK
                    / C:REJECT
```

This class contains the following server methods:

- File.Consume - start a queue consumer (ID = 1) (sync request)
- File.Cancel - end a queue consumer (ID = 3) (sync request)
- File.Open - request to start staging (ID = 5) (sync request)
- File.Open-Ok - confirm staging ready (ID = 6) (sync request)
- File.Stage - stage message content (ID = 7) (sync reply for Open-Ok, carries content)
- File.Publish - publish a message (ID = 8) (async)
- File.Ack - acknowledge one or more messages (ID = 10) (async)
- File.Reject - reject an incoming message (ID = 11) (async)

This class contains the following client methods:

- File.Consume-Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- File.Cancel-Ok - confirm a cancelled consumer (ID = 4) (sync reply for Cancel)
- File.Open - request to start staging (ID = 5) (sync request)
- File.Open-Ok - confirm staging ready (ID = 6) (sync request)
- File.Stage - stage message content (ID = 7) (sync reply for Open-Ok, carries content)
- File.Deliver - notify the client of a consumer message (ID = 9) (async)

Guidelines for implementors:

- The server MAY implement this class. Each method may have specific guidelines.
- The client MAY implement this class. Each method may have specific guidelines.
- Any assertion failures in the File methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.7.1 The File.Consume Method

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were created on, or until the client cancels them.

The File.Consume method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.

- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue to consume from.
- prefetch_size (short) - The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. May be set to zero, meaning "no specific limit". Note that other prefetch limits may still apply.
- prefetch_count (short) - Specifies a prefetch window in terms of whole messages. This is compatible with some file API implementations. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.
- no_local (bit) - If this field is set the server will not send messages to the client that published them.
- auto_ack (bit) - If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.
- exclusive (bit) - Request exclusive consumer access. If the server cannot grant this - because there are other consumers active - it raises a channel exception. The server MUST grant clients exclusive access to a queue if they ask for it.

This is the File.Consume pseudo-structure:

```
define File.Consume {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    prefetch_size      short       #  prefetch window in octets
    prefetch_count     short       #  prefetch window in messages
    no_local           bit         #  do not receive own messages
    auto_ack           bit         #  no acknowledgement needed
    exclusive          bit         #  request exclusive access
}
```

Guidelines for implementors:

- The server MAY restrict the number of consumers per channel to an arbitrary value, which MUST be at least 8, and MUST be specified in the Connection.Tune method.
- The client MUST be able to work with the server-defined limits with respect to the maximum number of consumers per channel.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: File.Consume-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.

- The server MUST grant clients exclusive access to a queue if they ask for it.

4.7.2 The File.Consume-Ok Method

This method provides the client with a consumer tag which it MUST use in methods that work with the consumer.

The File.Consume-Ok method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the File.Consume-Ok pseudo-structure:

```
define File.Consume-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.7.3 The File.Cancel Method

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

The File.Cancel method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the File.Cancel pseudo-structure:

```
define File.Cancel {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: File.Cancel-Ok unless there is an exception.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.7.4 The File.Cancel-Ok Method

This method confirms that the cancellation was completed. This method has no fields apart from the standard method header.

This is the File.Cancel-Ok pseudo-structure:

```
define File.Cancel-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.7.5 The File.Open Method

This method requests permission to start staging a message. Staging means sending the message into a temporary area at the recipient end and then delivering the message by referring to this temporary area. Staging is how the protocol handles partial file transfers - if a message is partially staged and the connection breaks, the next time the sender starts to stage it, it can restart from where it left off.

The File.Open method has the following specific fields:

- identifier (shortstr) - This is the staging identifier. This is an arbitrary string chosen by the sender. For staging to work correctly the sender must use the same staging identifier when staging the same message a second time after recovery from a failure. A good choice for the staging identifier would be the SHA1 hash of the message properties data (including the original filename, revised time, etc.).
- content_size (longlong) - The size of the content in octets. The recipient may use this information to allocate or check available space in advance, to avoid "disk full" errors during staging of very large messages. The sender MUST accurately fill this field. Zero-length content is permitted.

This is the File.Open pseudo-structure:

```
define File.Open {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    identifier         shortstr    #  staging identifier
    content_size       longlong    #  message content size
}
```

Guidelines for implementors:

- The server MUST implement this method.
- The client MUST implement this method.
- This method is a synchronous request that expects one of: File.Open-Ok unless there is an exception.
- The sender MUST accurately fill this field. Zero-length content is permitted.

4.7.6 The File.Open-Ok Method

This method confirms that the recipient is ready to accept staged data. If the message was already partially-staged at a previous time the recipient will report the number of octets already staged.

The File.Open-Ok method has the following specific fields:

- staged_size (longlong) - The amount of previously-staged content in octets. For a new message this will be zero. The sender MUST start sending data from this octet offset in the message, counting from zero. The recipient MAY decide how long to hold partially-staged content and MAY implement staging by always discarding partially-staged content. However if it uses the file content type it MUST support the staging methods.

This is the File.Open-Ok pseudo-structure:

```
define File.Open-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    staged_size        longlong    #  already staged amount
}
```

Guidelines for implementors:

- The server MUST implement this method.
- The client MUST implement this method.
- This method is a synchronous request that expects one of: File.Stage unless there is an exception.
- The sender MUST start sending data from this octet offset in the message, counting from zero.
- The recipient MAY decide how long to hold partially-staged content and MAY implement staging by always discarding partially-staged content. However if it uses the file content type it MUST support the staging methods.

4.7.7 The File.Stage Method

This method stages the message, sending the message content to the recipient from the octet offset specified in the Open-Ok method. This method has no fields apart from the standard method header.

This is the File.Stage pseudo-structure:

```
define File.Stage {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.7.8 The File.Publish Method

This method publishes a message to a specific proxy. The message will be forwarded to all registered queues and distributed to any active consumers when the transaction is committed.

The File.Publish method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy to publish to. If the proxy does not exist the server will raise a channel exception.
- immediate (bit) - Asserts that the message is delivered to one or more consumers immediately and causes a channel exception if this is not the case.
- identifier (shortstr) - This is the staging identifier of the message to publish. The message must have been staged. Note that a client can send the Publish method asynchronously without waiting for staging to finish.

This is the File.Publish pseudo-structure:

```
define File.Publish {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    proxy              shortstr    #  proxy name
    immediate          bit         #  assert immediate delivery
    identifier         shortstr    #  staging identifier
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- The "proxy" field MUST not be empty.

4.7.9 The File.Deliver Method

This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

The File.Deliver method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- redelivered (bit) - This indicates that the message has been previously delivered to this or another client. The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.

- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy that the message was originally published to.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue that the message came from. Note that a single channel can start many consumers on different queues.
- identifier (shortstr) - This is the staging identifier of the message to deliver. The message must have been staged. Note that a server can send the Deliver method asynchronously without waiting for staging to finish.

This is the File.Deliver pseudo-structure:

```
define File.Deliver {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    redelivered        bit         #  signal a redelivered message
    proxy              shortstr    #  proxy name
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    identifier         shortstr    #  staging identifier
}
```

Guidelines for implementors:

- The server SHOULD track the number of times a message has been delivered to clients and when a message is redelivered a certain number of times - e.g. 5 times - without being acknowledged, the server SHOULD consider the message to be unprocessable (possibly causing client applications to abort), and move the message to a dead letter queue.
- The client MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.
- The ”proxy” field MUST not be empty.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The ”queue” field MUST not be empty.

4.7.10 The File.Ack Method

This method acknowledges one or more messages delivered via the Deliver method. The client can ask to confirm a single message or a set of messages up to and including a specific message.

The File.Ack method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- multiple (bit) - If set to 1, the delivery tag is treated as "up to and including", so that the client can acknowledge multiple messages with a single method. If set to zero, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, tells the server to acknowledge all outstanding mesages. The server MUST validate that a non-zero delivery-tag refers to an delivered message, and raise a channel exception if this is not the case.

This is the File.Ack pseudo-structure:

```
define File.Ack {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    multiple           bit         #  acknowledge multiple messages
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The server MUST validate that a non-zero delivery-tag refers to an delivered message, and raise a channel exception if this is not the case.

4.7.11 The File.Reject Method

This method allows a client to reject a message. It can be used to return untreatable messages to their original queue. Note that file content is staged before delivery, so the client will not use this method to interrupt delivery of a large message.

The File.Reject method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- requeue (bit) - If this field is zero, the message will be discarded. If this bit is 1, the server will attempt to requeue the message. The server MUST NOT deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

This is the File.Reject pseudo-structure:

```
define File.Reject {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    requeue            bit         #  requeue the message
}
```

Guidelines for implementors:

- The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.
- A client MUST NOT use this method as a means of selecting messages to process. A rejected message MAY be discarded or dead-lettered, not necessarily passed to another client.
- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The server MUST NOT deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

4.8 The Stream Class

The stream class provides methods that support multimedia streaming. The stream class uses the following semantics: one message is one packet of data; delivery is unacknowleged and unreliable; the consumer can specify quality of service parameters that the server can try to adhere to; lower-priority messages may be discarded in favour of high priority messages. The ID of the Stream class is 8.

This is the formal grammar for the class:

```
stream              = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:PUBLISH content
                    / S:DELIVER content
```

This class contains the following server methods:

- Stream.Consume - start a queue consumer (ID = 1) (sync request)
- Stream.Cancel - end a queue consumer (ID = 3) (sync request)
- Stream.Publish - publish a message (ID = 5) (async, carries content)

This class contains the following client methods:

- Stream.Consume-Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- Stream.Cancel-Ok - confirm a cancelled consumer (ID = 4) (sync reply for Cancel)
- Stream.Deliver - notify the client of a consumer message (ID = 6) (async, carries content)

Guidelines for implementors:

- The server SHOULD implement this class. Each method may have specific guidelines.
- The client MAY implement this class. Each method may have specific guidelines.
- Any assertion failures in the Stream methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.8.1 The Stream.Consume Method

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were created on, or until the client cancels them.

The Stream.Consume method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue to consume from.
- prefetch_size (short) - The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. May be set to zero, meaning "no specific limit". Note that other prefetch limits may still apply.
- prefetch_count (short) - Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it.
- consume_rate (long) - Specifies a desired transfer rate in octets per second. This is usually determined by the application that uses the streaming data. A value of zero means "no limit", i.e. as rapidly as possible. The server MAY ignore the prefetch values and consume rates, depending on the type of stream and the ability of the server to queue and/or reply it. The server MAY drop low-priority messages in favour of high-priority messages.
- no_local (bit) - If this field is set the server will not send messages to the client that published them.
- exclusive (bit) - Request exclusive consumer access. If the server cannot grant this - because there are other consumers active - it raises a channel exception. The server MUST grant clients exclusive access to a queue or subscription if they ask for it.

This is the Stream.Consume pseudo-structure:

```
define Stream.Consume {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    ticket             short       #  access ticket granted by server
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
    prefetch_size      short       #  prefetch window in octets
    prefetch_count     short       #  prefetch window in messages
    consume_rate       long        #  transfer rate in octets/second
    no_local           bit         #  do not receive own messages
    exclusive          bit         #  request exclusive access
}
```

Guidelines for implementors:

- The server MAY restrict the number of consumers per channel to an arbitrary value, which MUST be at least 8, and MUST be specified in the Connection.Tune method.
- The client MUST be able to work with the server-defined limits with respect to the maximum number of consumers per channel.
- Streaming applications SHOULD use different channels to select different streaming resolutions. AMQP/Fast makes no provision for filtering and/or transforming streams except on the basis of priority-based selective delivery of individual messages.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Stream.Consume-Ok unless there is an exception.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The "queue" field MUST not be empty.
- The server MAY ignore the prefetch values and consume rates, depending on the type of stream and the ability of the server to queue and/or reply it. The server MAY drop low-priority messages in favour of high-priority messages.
- The server MUST grant clients exclusive access to a queue or subscription if they ask for it.

4.8.2 The Stream.Consume-Ok Method

This method provides the client with a consumer tag which it may use in methods that work with the consumer.

The Stream.Consume-Ok method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the Stream.Consume-Ok pseudo-structure:

```
define Stream.Consume-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.8.3 The Stream.Cancel Method

This method cancels a consumer. Since message delivery is asynchronous the client may continue to receive messages for a short while after cancelling a consumer. It may process or discard these as appropriate.

The Stream.Cancel method has the following specific fields:

- consumer_tag (short) - The server-assigned and channel-specific consumer tag This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

This is the Stream.Cancel pseudo-structure:

```
define Stream.Cancel {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    consumer_tag       short       #  server-assigned consumer tag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Stream.Cancel-Ok unless there is an exception.
- This field is an arbitrary ID number for the consumer. The consumer tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the consumer was created.

4.8.4 The Stream.Cancel-Ok Method

This method confirms that the cancellation was completed. This method has no fields apart from the standard method header.

This is the Stream.Cancel-Ok pseudo-structure:

```
define Stream.Cancel-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.8.5 The Stream.Publish Method

This method publishes a message to a specific proxy. The message will be forwarded to all registered queues and distributed to any active consumers when the transaction is committed.

The Stream.Publish method has the following specific fields:

- ticket (short) - An access ticket granted by the server for a certain set of access rights within a specific realm. Access tickets may be shared across channels within a connection and expire with the connection. The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy to publish to. If the proxy does not exist the server will raise a channel exception.
- immediate (bit) - Asserts that the message is delivered to one or more consumers immediately and causes a channel exception if this is not the case.

This is the Stream.Publish pseudo-structure:

```
define Stream.Publish {
    method_class       octet      #  class ID
    method_id          octet      #  method ID
    method_flags       octet      #  method flags
    method_synctag     long       #  synctag
    ticket             short      #  access ticket granted by server
    proxy              shortstr   #  proxy name
    immediate          bit        #  assert immediate delivery
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- The client MUST provide a valid access ticket giving "write" access rights to the access realm for the proxy.
- The "proxy" field MUST not be empty.

4.8.6 The Stream.Deliver Method

This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

The Stream.Deliver method has the following specific fields:

- delivery_tag (longlong) - The server-assigned and channel-specific delivery tag This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- redelivered (bit) - This indicates that the message has been previously delivered to this or another client. The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.

- proxy (shortstr) - The proy name identifies the proxy within the virtual host. A proxy name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Proxy names are case-sensitive. Specifies the name of the proxy that the message was originally published to.
- namespace (shortstr) - The queue namespace is an arbitrary string chosen by the application. The combination of queue namespace and queue name is unique per virtual host. A queue namespace consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue namespaces are case-sensitive. The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- queue (shortstr) - The queue name identifies the queue within the namespace and the virtual host. A queue name consists of any combination of at least one of [A-Za-z0-9] plus zero or more of [.-_/+!=:]. Queue names starting with _ are reserved for server use. Queue names are case-sensitive. Specifies the name of the queue that the message came from. Note that a single channel can start many consumers on different queues.

This is the Stream.Deliver pseudo-structure:

```
define Stream.Deliver {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    delivery_tag       longlong    #  server-assigned delivery tag
    redelivered        bit         #  signal a redelivered message
    proxy              shortstr    #  proxy name
    namespace          shortstr    #  queue namespace
    queue              shortstr    #  queue name
}
```

Guidelines for implementors:

- The server SHOULD track the number of times a message has been delivered to clients and when a message is redelivered a certain number of times - e.g. 5 times - without being acknowledged, the server SHOULD consider the message to be unprocessable (possibly causing client applications to abort), and move the message to a dead letter queue.
- The client MUST implement this method.
- This method is asynchronous. The recipient MUST handle it at any point as defined by the class grammar. The sender MUST not expect any specific reply unless the SYNCHREQ flag is set (which will solicit a Channel.Synch reply).
- This field is an sequential ID number for the delivery. The delivery tag is specific to a single channel and the client MUST NOT use this tag in any channel except the channel where the message was delivered.
- The client MUST NOT rely on this field but MUST take it as a hint that the message may already have been processed. A fully robust client must be able to track duplicate received messages on non transacted, and locally-transacted channels. The server SHOULD try to signal redelivered messages when it can. When redelivering a message that was not successfully acknowledged, the server SHOULD deliver it to the original client if possible.
- The ”proxy” field MUST not be empty.
- The queue namespace MAY be empty - the empty namespace acts just like any other name space.
- The ”queue” field MUST not be empty.

4.9 The Tx Class

Standard transactions provide so-called "1.5 phase commit". We can ensure that work is never lost, but there is a chance of confirmations being lost, so that messages may be resent. Applications that use standard transactions must be able to detect and ignore duplicate messages. The ID of the Tx class is 9.

This is the formal grammar for the class:

```
tx                = C:COMMIT S:COMMIT-OK
                  / C:ABORT S:ABORT-OK
```

This class contains the following server methods:

- Tx.Commit - commit the current transaction (ID = 1) (sync request)
- Tx.Abort - abandon the current transaction (ID = 3) (sync request)

This class contains the following client methods:

- Tx.Commit-Ok - confirm a successful commit (ID = 2) (sync reply for Commit)
- Tx.Abort-Ok - confirm a successful abort (ID = 4) (sync reply for Abort)

Guidelines for implementors:

- An client using standard transactions SHOULD be able to track all messages received within a reasonable period, and thus detect and reject duplicates of the same message. It SHOULD NOT pass these to the application layer.
- The server SHOULD implement this class. Each method may have specific guidelines.
- The client MAY implement this class. Each method may have specific guidelines.
- Any assertion failures in the Tx methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.9.1 The Tx.Commit Method

This method commits all messages published and acknowledged in the current transaction. A new transaction starts immediately after a commit. This method has no fields apart from the standard method header.

This is the Tx.Commit pseudo-structure:

```
define Tx.Commit {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Tx.Commit-Ok unless there is an exception.

4.9.2 The Tx.Commit-Ok Method

This method confirms to the client that the commit succeeded. Note that if a commit fails, the server raises a channel exception. This method has no fields apart from the standard method header.

This is the Tx.Commit-Ok pseudo-structure:

```
define Tx.Commit-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.9.3 The Tx.Abort Method

This method abandons all messages published and acknowledged in the current transaction. A new transaction starts immediately after a rollback. This method has no fields apart from the standard method header.

This is the Tx.Abort pseudo-structure:

```
define Tx.Abort {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The server MUST implement this method.
- This method is a synchronous request that expects one of: Tx.Abort-Ok unless there is an exception.

4.9.4 The Tx.Abort-Ok Method

This method confirms to the client that the abort succeeded. Note that if an abort fails, the server raises a channel exception. This method has no fields apart from the standard method header.

This is the Tx.Abort-Ok pseudo-structure:

```
define Tx.Abort-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.10 The Dtx Class

Distributed transactions provide so-called "2-phase commit". This is slower and more complex than standard transactions but provides more assurance that messages will be delivered exactly once. The AMQP/Fast distributed transaction model supports the X-Open XA architecture and other distributed transaction implementations. The Dtx class assumes that the server has a private communications channel (not AMQP/Fast) to a distributed transaction coordinator. The ID of the Dtx class is 10.

This is the formal grammar for the class:

```
dtx                  = C:START S:START-OK
```

This class contains the following server methods:

- Dtx.Start - start a new distributed transaction (ID = 1) (sync request)

This class contains the following client methods:

- Dtx.Start-Ok - confirm the start of a new distributed transaction (ID = 2) (sync reply for Start)

Guidelines for implementors:

- The server MAY implement this class. Each method may have specific guidelines.
- The client MAY implement this class. Each method may have specific guidelines.
- Any assertion failures in the Dtx methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.10.1 The Dtx.Start Method

This method starts a new distributed transaction. This must be the first method on a new channel that uses the distributed transaction mode, before any methods that publish or consume messages.

The Dtx.Start method has the following specific fields:

- dtx_identifier (shortstr) - The distributed transaction key. This identifies the transaction so that the AMQP/Fast server can coordinate with the distributed transaction coordinator.

This is the Dtx.Start pseudo-structure:

```
define Dtx.Start {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    dtx_identifier     shortstr    #  distributed transaction identifier
}
```

Guidelines for implementors:

- The server MAY implement this method.
- This method is a synchronous request that expects one of: Dtx.Start-Ok unless there is an exception.
- The "dtx_identifier" field MUST not be empty.

4.10.2 The Dtx.Start-Ok Method

This method confirms to the client that the transaction started. Note that if a start fails, the server raises a channel exception. This method has no fields apart from the standard method header.

This is the Dtx.Start-Ok pseudo-structure:

```
define Dtx.Start-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.11 The Test Class

The test class provides methods for a peer to test the basic operational correctness of another peer. The test methods are intended to ensure that all peers respect at least the basic elements of the protocol, such as frame and content organisation and field types. We assume that a specially-designed peer, a "monitor client" would perform such tests. The ID of the Test class is 11.

This is the formal grammar for the class:

```
test               = C:INTEGER S:INTEGER-OK
                   / S:INTEGER C:INTEGER-OK
                   / C:STRING S:STRING-OK
                   / S:STRING C:STRING-OK
                   / C:TABLE S:TABLE-OK
                   / S:TABLE C:TABLE-OK
                   / C:CONTENT S:CONTENT-OK
                   / S:CONTENT C:CONTENT-OK
```

This class contains the following server methods:

- Test.Integer - test integer handling (ID = 1) (sync request)
- Test.Integer-Ok - report integer test result (ID = 2) (sync reply for Integer)
- Test.String - test string handling (ID = 3) (sync request)
- Test.String-Ok - report string test result (ID = 4) (sync reply for String)
- Test.Table - test field table handling (ID = 5) (sync request)
- Test.Table-Ok - report table test result (ID = 6) (sync reply for Table)
- Test.Content - test content handling (ID = 7) (sync request, carries content)
- Test.Content-Ok - report content test result (ID = 8) (sync reply for Content, carries content)

This class contains the following client methods:

- Test.Integer - test integer handling (ID = 1) (sync request)
- Test.Integer-Ok - report integer test result (ID = 2) (sync reply for Integer)
- Test.String - test string handling (ID = 3) (sync request)
- Test.String-Ok - report string test result (ID = 4) (sync reply for String)
- Test.Table - test field table handling (ID = 5) (sync request)
- Test.Table-Ok - report table test result (ID = 6) (sync reply for Table)
- Test.Content - test content handling (ID = 7) (sync request, carries content)
- Test.Content-Ok - report content test result (ID = 8) (sync reply for Content, carries content)

Guidelines for implementors:

- The server MUST implement this class. Each method may have specific guidelines.
- The client SHOULD implement this class. Each method may have specific guidelines.
- Any assertion failures in the Test methods MUST BE treated as channel exceptions - i.e. the peer that detects the error MUST respond with Channel.Close.

4.11.1 The Test.Integer Method

This method tests the peer's capability to correctly marshal integer data.

The Test.Integer method has the following specific fields:

- integer_1 (octet) - An octet integer test value.
- integer_2 (short) - A short integer test value.
- integer_3 (long) - A long integer test value.
- integer_4 (longlong) - A long long integer test value.
- operation (octet) - The client must execute this operation on the provided integer test fields and return the result.

This is the Test.Integer pseudo-structure:

```
define Test.Integer {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    integer_1          octet       #  octet test value
    integer_2          short       #  short test value
    integer_3          long        #  long test value
    integer_4          longlong    #  long-long test value
    operation          octet       #  operation to test
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Test.Integer-Ok unless there is an exception.
- The "operation" field MUST be one of: 1=add (value.), 2=min (value.), 3=max (value.).

4.11.2 The Test.Integer-Ok Method

This method reports the result of an Integer method.

The Test.Integer-Ok method has the following specific fields:

- result (longlong) - The result of the tested operation.

This is the Test.Integer-Ok pseudo-structure:

```
define Test.Integer-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    result             longlong    #  result value
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.11.3 The Test.String Method

This method tests the peer's capability to correctly marshal string data.

The Test.String method has the following specific fields:

- string_1 (shortstr) - An short string test value.
- string_2 (longstr) - A long string test value.
- operation (octet) - The client must execute this operation on the provided string test fields and return the result.

This is the Test.String pseudo-structure:

```
define Test.String {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    string_1           shortstr    #  short string test value
    string_2           longstr     #  long string test value
    operation          octet       #  operation to test
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Test.String-Ok unless there is an exception.
- The "operation" field MUST be one of: 1=add (value.), 2=min (value.), 3=max (value.).

4.11.4 The Test.String-Ok Method

This method reports the result of a String method.

The Test.String-Ok method has the following specific fields:

- result (longstr) - The result of the tested operation.

This is the Test.String-Ok pseudo-structure:

```
define Test.String-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    result             longstr     #  result value
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.

- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.11.5 The Test.Table Method

This method tests the peer's capability to correctly marshal field table data.

The Test.Table method has the following specific fields:

- table (table) - A field table of test values.
- integer_op (octet) - The client must execute this operation on the provided field table integer values and return the result.
- string_op (octet) - The client must execute this operation on the provided field table string values and return the result.

This is the Test.Table pseudo-structure:

```
define Test.Table {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    table              table       #  field table of test values
    integer_op         octet       #  operation to test on integers
    string_op          octet       #  operation to test on strings
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Test.Table-Ok unless there is an exception.
- The "integer_op" field MUST be one of: 1=add (value.), 2=min (value.), 3=max (value.).
- The "string_op" field MUST be one of: 1=add (value.), 2=min (value.), 3=max (value.).

4.11.6 The Test.Table-Ok Method

This method reports the result of a Table method.

The Test.Table-Ok method has the following specific fields:

- integer_result (longlong) - The result of the tested integer operation.
- string_result (longstr) - The result of the tested string operation.

This is the Test.Table-Ok pseudo-structure:

```
define Test.Table-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    integer_result     longlong    #  integer result value
    string_result      longstr     #  string result value
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.

- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.

4.11.7 The Test.Content Method

This method tests the peer's capability to correctly marshal content. This method has no fields apart from the standard method header.

This is the Test.Content pseudo-structure:

```
define Test.Content {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous request that expects one of: Test.Content-Ok unless there is an exception.

4.11.8 The Test.Content-Ok Method

This method reports the result of a Content method. It contains the content checksum and echoes the original content as provided.

The Test.Content-Ok method has the following specific fields:

- content_checksum (long) - The 32-bit checksum of the content, calculated by adding the content into a 32-bit accumulator.

This is the Test.Content-Ok pseudo-structure:

```
define Test.Content-Ok {
    method_class       octet       #  class ID
    method_id          octet       #  method ID
    method_flags       octet       #  method flags
    method_synctag     long        #  synctag
    content_checksum   long        #  content hash
}
```

Guidelines for implementors:

- The client MUST implement this method.
- The server MUST implement this method.
- This method is a synchronous response. The sender MUST NOT expect a specific method in return, except as defined by the class grammar.