Copyright Notice

License

The following disclaimers, which you hereby also acknowledge as to any use you may make of the Advanced Messaging Queue Protocol Specification:

THE ADVANCED MESSAGING QUEUE PROTOCOL SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE ADVANCED MESSAGING QUEUE PROTOCOL SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF THE ADVANCED MESSAGING QUEUE PROTOCOL SPECIFICATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE, IMPLEMENTATION OR DISTRIBUTION OF THE ADVANCED MESSAGING QUEUE PROTOCOL SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Advanced Messaging Queue Protocol Specification or its contents without specific, written prior permission. Title to copyright in the Advanced Messaging Queue Protocol Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Upon termination of your license or rights under this Agreement, you shall destroy all copies of the Advanced Messaging Queue Protocol Specification in your possession or control.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

1 AMQ Protocol (major=10, minor=3)

1.1 Class and Method Ids

These are the AMQP class and method ids. Note that these may change in new versions of AMQP and implementers are strongly recommended to use the AMQP class specifications as a source for the class and method ids rather than hard-coding these values.

These are the ID values for each class:

```
Connection = 10
Channel = 20
Access = 30
Exchange = 40
Queue = 50
Basic = 60
File = 70
Stream = 80
Tx = 90
Dtx = 100
Tunnel = 110
Test = 120
Cluster = 61440
```

These are the ID values for the Connection methods:

```
Connection.Start = 10
Connection.Start-Ok = 20
Connection.Secure = 30
Connection.Secure-Ok = 40
Connection.Tune = 50
Connection.Tune-Ok = 60
Connection.Open = 70
Connection.Open-Ok = 80
Connection.Redirect = 90
Connection.Close = 100
Connection.Close-Ok = 110
```

These are the ID values for the Channel methods:

```
Channel.Open = 10
Channel.Open-Ok = 20
Channel.Flow = 30
Channel.Flow-Ok = 40
Channel.Alert = 50
Channel.Close = 60
Channel.Close-Ok = 70
```

These are the ID values for the Access methods:

```
Access.Request = 10
Access.Request-Ok = 20
```

These are the ID values for the Exchange methods:

```
Exchange.Declare = 10
Exchange.Declare-Ok = 20
Exchange.Delete = 30
Exchange.Delete-Ok = 40
```

These are the ID values for the Queue methods:

```
Queue.Declare = 10
Queue.Declare-Ok = 20
Queue.Bind = 30
Queue.Bind-Ok = 40
Queue.Purge = 50
Queue.Purge-Ok = 60
Queue.Delete = 70
Queue.Delete-Ok = 80
```

These are the ID values for the Basic methods:

```
Basic.Consume = 10
Basic.Consume-Ok = 20
Basic.Cancel = 30
Basic.Cancel-Ok = 40
Basic.Publish = 50
Basic.Return = 60
Basic.Deliver = 70
Basic.Get = 80
Basic.Get-Ok = 90
Basic.Get-Empty = 100
Basic.Ack = 110
Basic.Reject = 120
```

These are the ID values for the File methods:

```
File.Consume = 10
File.Consume-Ok = 20
File.Cancel = 30
File.Cancel-Ok = 40
File.Open = 50
File.Open-Ok = 60
File.Stage = 70
File.Publish = 80
File.Return = 90
File.Deliver = 100
File.Ack = 110
File.Reject = 120
```

These are the ID values for the Stream methods:

```
Stream.Consume = 10
Stream.Consume-Ok = 20
Stream.Cancel = 30
Stream.Cancel-Ok = 40
Stream.Publish = 50
Stream.Return = 60
Stream.Deliver = 70
```

These are the ID values for the Tx methods:

```
Tx.Select = 10
Tx.Select-Ok = 20
Tx.Commit = 30
Tx.Commit-Ok = 40
Tx.Rollback = 50
Tx.Rollback-Ok = 60
```

These are the ID values for the Dtx methods:

```
Dtx.Select = 10
Dtx.Select-Ok = 20
Dtx.Start = 30
Dtx.Start-Ok = 40
```

These are the ID values for the Tunnel methods:

```
Tunnel.Request = 10
```

These are the ID values for the Test methods:

```
Test.Integer = 10
Test.Integer-Ok = 20
Test.String = 30
Test.String-Ok = 40
Test.Table = 50
Test.Table-Ok = 60
Test.Content = 70
Test.Content-Ok = 80
```

These are the ID values for the Cluster methods:

```
Cluster.Hello = 10
Cluster.Status = 20
Cluster.Bind = 30
```

1.1.1 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 10.

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 | S:REDIRECT
    challenge           = S:SECURE C:SECURE-OK
    use-connection      = *channel
    close-connection    = C:CLOSE S:CLOSE-OK
                        / S:CLOSE C:CLOSE-OK
```

The server accepts the following methods:

- Connection.Start-Ok (ID=20) - select security mechanism and locale : Sync response to Start , carries content
- Connection.Secure-Ok (ID=40) - security mechanism response : Sync response to Secure
- Connection.Tune-Ok (ID=60) - negotiate connection tuning parameters : Sync response to Tune , carries content
- Connection.Open (ID=70) - open connection to virtual host : Sync request , carries content
- Connection.Close (ID=100) - request a connection close : Sync request , carries content
- Connection.Close-Ok (ID=110) - confirm a connection close : Sync response to Close

The client accepts the following methods:

- Connection.Start (ID=10) - start connection negotiation : Sync request , carries content
- Connection.Secure (ID=30) - security mechanism challenge : Sync request
- Connection.Tune (ID=50) - propose connection tuning parameters : Sync request , carries content
- Connection.Open-Ok (ID=80) - signal that the connection is ready : Sync response to Open
- Connection.Redirect (ID=90) - asks the client to use a different server : Sync response to Open , carries content
- Connection.Close (ID=100) - request a connection close : Sync request , carries content
- Connection.Close-Ok (ID=110) - confirm a connection close : Sync response to Close

1.1.1.1 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 Start method has the following specific fields:

This is the Start pseudo-structure:

Guidelines for implementers:

- 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.
- All servers MUST support at least the en_US locale.

1.1.1.2 The Connection.Start-Ok Method

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

The Start-Ok method has the following specific fields:

This is the Start-Ok pseudo-structure:

1.1.1.3 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 Secure method has the following specific fields:

This is the Secure pseudo-structure:

1.1.1.4 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 Secure-Ok method has the following specific fields:

This is the Secure-Ok pseudo-structure:

1.1.1.5 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 Tune method has the following specific fields:

This is the Tune pseudo-structure:

1.1.1.6 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 Tune-Ok method has the following specific fields:

This is the Tune-Ok pseudo-structure:

1.1.1.7 The Connection.Open Method

This method opens a connection to a virtual host, which is a collection of resources, and acts to separate multiple application domains within a server.

The Open method has the following specific fields:

This is the Open pseudo-structure:

Guidelines for implementers:

- The client MUST open the context before doing any work on the connection.
- If the server supports multiple virtual hosts, it MUST enforce a full separation of exchanges, queues, and all associated entities per virtual host. An application, connected to a specific virtual host, MUST NOT be able to access resources of another virtual host.
- The server SHOULD verify that the client has permission to access the specified virtual host.
- The server MAY configure arbitrary limits per virtual host, such as the number of each type of entity that may be used, per connection and/or in total.
- When the client uses the insist option, the server SHOULD accept the client connection unless it is technically unable to do so.

1.1.1.8 The Connection.Open-Ok Method

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

The Open-Ok method has the following specific fields:

This is the Open-Ok pseudo-structure:

1.1.1.9 The Connection.Redirect Method

This method redirects the client to another server, based on the requested virtual host and/or capabilities.

The Redirect method has the following specific fields:

This is the Redirect pseudo-structure:

Guidelines for implementers:

- When getting the Connection.Redirect method, the client SHOULD reconnect to the host specified, and if that host is not present, to any of the hosts specified in the known-hosts list.

1.1.1.10 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 and method id of the method which caused the exception.

The Close method has the following specific fields:

This is the Close pseudo-structure:

Guidelines for implementers:

- After sending this method any received method except the Close-OK method 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 the Close-OK method.
- When a server receives the Close method from a client it MUST delete all server-side resources associated with the client's context. A client CANNOT reconnect to a context after sending or receiving a Close method.

1.1.1.11 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.

The Close-Ok method has the following specific fields:

This is the Close-Ok pseudo-structure:

Guidelines for implementers:

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

1.1.2 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 20.

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-OK
                    / S:FLOW C:FLOW-OK
                    / S:ALERT
                    / functional-class
close-channel       = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

The server accepts the following methods:

- Channel.Open (ID=10) - open a channel for use : Sync request , carries content
- Channel.Flow (ID=30) - enable/disable flow from peer : Sync request
- Channel.Flow-Ok (ID=40) - confirm a flow method : Async
- Channel.Close (ID=60) - request a channel close : Sync request , carries content
- Channel.Close-Ok (ID=70) - confirm a channel close : Sync response to Close

The client accepts the following methods:

- Channel.Open-Ok (ID=20) - signal that the channel is ready : Sync response to Open
- Channel.Flow (ID=30) - enable/disable flow from peer : Sync request
- Channel.Flow-Ok (ID=40) - confirm a flow method : Async
- Channel.Alert (ID=50) - send a non-fatal warning message : Async , carries content
- Channel.Close (ID=60) - request a channel close : Sync request , carries content
- Channel.Close-Ok (ID=70) - confirm a channel close : Sync response to Close

1.1.2.1 The Channel.Open Method

This method opens a virtual connection (a channel).

The Open method has the following specific fields:

This is the Open pseudo-structure:

Guidelines for implementers:

- This method MUST NOT be called when the channel is already open.
- The server MUST NOT send a client more data in advance than this value allows. If sending specific content in advance would exhaust the channel prefetch window, it MUST NOT send the content. Setting this field to a very low non-zero value (such as 1) effectively disables all prefetching on the channel.

1.1.2.2 The Channel.Open-Ok Method

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

The Open-Ok method has the following specific fields:

This is the Open-Ok pseudo-structure:

1.1.2.3 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 peer that receives a request to stop sending content should finish sending the current content, if any, and then wait until it receives a Flow restart method.

The Flow method has the following specific fields:

This is the Flow pseudo-structure:

Guidelines for implementers:

- When a new channel is opened, it is active. Some applications assume that channels are inactive until started. To emulate this behaviour a client MAY open the channel, then pause it.
- 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 exchangeing 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.

1.1.2.4 The Channel.Flow-Ok Method

Confirms to the peer that a flow command was received and processed.

The Flow-Ok method has the following specific fields:

This is the Flow-Ok pseudo-structure:

1.1.2.5 The Channel.Alert Method

This method allows the server to send a non-fatal warning to the client. This is used for methods that are normally asynchronous and thus do not have confirmations, and for which the server may detect errors that need to be reported. Fatal errors are handled as channel or connection exceptions; non-fatal errors are sent through this method.

The Alert method has the following specific fields:

This is the Alert pseudo-structure:

1.1.2.6 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 and method id of the method which caused the exception.

The Close method has the following specific fields:

This is the Close pseudo-structure:

Guidelines for implementers:

- 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..

1.1.2.7 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.

The Close-Ok method has the following specific fields:

This is the Close-Ok pseudo-structure:

Guidelines for implementers:

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

1.1.3 The Access Class

The protocol control 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 30.

This is the formal grammar for the class:

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

The server accepts the following methods:

- Access.Request (ID=10) - request an access ticket : Sync request , carries content

The client accepts the following methods:

- Access.Request-Ok (ID=20) - grant access to server resources : Sync response to Request

1.1.3.1 The Access.Request 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 are a per-channel resource.

The Request method has the following specific fields:

This is the Request pseudo-structure:

Guidelines for implementers:

- The realm name MUST start with either "/data" (for application resources) or "/admin" (for server administration resources). If the realm starts with any other path, the server MUST raise a connection exception with reply code 403 (access refused).
- 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.
- If the specified realm is not known to the server, the server must raise a channel exception with reply code 402 (invalid path).

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

This is the Request-Ok pseudo-structure:

Guidelines for implementers:

- 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.

1.1.4 The Exchange Class

Exchanges match and distribute messages across queues. Exchanges can be configured in the server or created at runtime. The ID of the Exchange Class is 40.

This is the formal grammar for the class:

```
exchange            = C:DECLARE  S:DECLARE-OK
                    / C:DELETE   S:DELETE-OK
```

The server accepts the following methods:

- Exchange.Declare (ID=10) - declare exchange, create if needed : Sync request , carries content
- Exchange.Delete (ID=30) - delete an exchange : Sync request , carries content

The client accepts the following methods:

- Exchange.Declare-Ok (ID=20) - confirms an exchange declaration : Sync response to Declare
- Exchange.Delete-Ok (ID=40) - confirm deletion of an exchange : Sync response to Delete

1.1.4.1 The Exchange.Declare Method

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

The Declare method has the following specific fields:

This is the Declare pseudo-structure:

Guidelines for implementers:

- amq_exchange_23 The server SHOULD support a minimum of 16 exchanges per virtual host and ideally, impose no limit except as defined by available resources.
- The client MUST provide a valid access ticket giving "active" access to the realm in which the exchange exists or will be created, or "passive" access if the if-exists flag is set.
- amq_exchange_15 Exchange names starting with "amq." are reserved for predeclared and standardised exchanges. If the client attempts to create an exchange starting with "amq.", the server MUST raise a channel exception with reply code 403 (access refused).
- amq_exchange_16 If the exchange already exists with a different type, the server MUST raise a connection exception with a reply code 507 (not allowed).
- amq_exchange_18 If the server does not support the requested exchange type it MUST raise a connection exception with a reply code 503 (command invalid).
- amq_exchange_05 If set, and the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).
- amq_exchange_24 The server MUST support both durable and transient exchanges.
- The server MUST ignore the durable field if the exchange already exists.
- amq_exchange_02 The server SHOULD allow for a reasonable delay between the point when it determines that an exchange is not being used (or no longer used), and the point when it deletes the

exchange. At the least it must allow a client to create an exchange and then bind a queue to it, with a small but non-zero delay between these two actions.

- amq_exchange_25 The server MUST ignore the auto-delete field if the exchange already exists.

1.1.4.2 The Exchange.Declare-Ok Method

This method confirms a Declare method and confirms the name of the exchange, essential for automatically-named exchanges.

The Declare-Ok method has the following specific fields:

This is the Declare-Ok pseudo-structure:

1.1.4.3 The Exchange.Delete Method

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

The Delete method has the following specific fields:

This is the Delete pseudo-structure:

1.1.4.4 The Exchange.Delete-Ok Method

This method confirms the deletion of an exchange.

The Delete-Ok method has the following specific fields:

This is the Delete-Ok pseudo-structure:

1.1.5 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 exchange in order to receive messages from publishers. The ID of the Queue Class is 50.

This is the formal grammar for the class:

```
queue               = C:DECLARE  S:DECLARE-OK
                    / C:BIND     S:BIND-OK
                    / C:PURGE    S:PURGE-OK
                    / C:DELETE   S:DELETE-OK
```

The server accepts the following methods:

- Queue.Declare (ID=10) - declare queue, create if needed : Sync request , carries content
- Queue.Bind (ID=30) - bind queue to an exchange : Sync request , carries content
- Queue.Purge (ID=50) - purge a queue : Sync request , carries content
- Queue.Delete (ID=70) - delete a queue : Sync request , carries content

The client accepts the following methods:

- Queue.Declare-Ok (ID=20) - confirms a queue definition : Sync response to Declare , carries content
- Queue.Bind-Ok (ID=40) - confirm bind successful : Sync response to Bind
- Queue.Purge-Ok (ID=60) - confirms a queue purge : Sync response to Purge
- Queue.Delete-Ok (ID=80) - confirm deletion of a queue : Sync response to Delete

1.1.5.1 The Queue.Declare 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 Declare method has the following specific fields:

This is the Declare pseudo-structure:

Guidelines for implementers:

- amq_queue_34 The server MUST create a default binding for a newly-created queue to the default exchange, which is an exchange of type 'direct'.
- amq_queue_35 The server SHOULD support a minimum of 256 queues per virtual host and ideally, impose no limit except as defined by available resources.
- amq_queue_10 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 Declare-Ok method.
- amq_queue_32 Queue names starting with "amq." are reserved for predeclared and standardised server queues. If the queue name starts with "amq." and the passive option is zero, the server MUST raise a connection exception with reply code 403 (access refused).
- amq_queue_05 If set, and the queue does not already exist, the server MUST respond with a reply code 404 (not found) and raise a channel exception.
- amq_queue_03 The server MUST recreate the durable queue after a restart.
- amq_queue_36 The server MUST support both durable and transient queues.
- amq_queue_37 The server MUST ignore the durable field if the queue already exists.
- amq_queue_38 The server MUST support both exclusive (private) and non-exclusive (shared) queues.
- amq_queue_04 The server MUST raise a channel exception if 'exclusive' is specified and the queue already exists and is owned by a different connection.

- amq_queue_02 The server SHOULD allow for a reasonable delay between the point when it determines that a queue is not being used (or no longer used), and the point when it deletes the queue. At the least it must allow a client to create a queue and then create a consumer to read from it, with a small but non-zero delay between these two actions. The server should equally allow for clients that may be disconnected prematurely, and wish to re-consume from the same queue without losing messages. We would recommend a configurable timeout, with a suitable default value being one minute.
- amq_queue_31 The server MUST ignore the auto-delete field if the queue already exists.

1.1.5.2 The Queue.Declare-Ok Method

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

The Declare-Ok method has the following specific fields:

This is the Declare-Ok pseudo-structure:

1.1.5.3 The Queue.Bind Method

This method binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and-forward queues are bound to a dest exchange and subscription queues are bound to a dest_wild exchange.

The Bind method has the following specific fields:

This is the Bind pseudo-structure:

Guidelines for implementers:

- amq_queue_25 A server MUST allow ignore duplicate bindings - that is, two or more bind methods for a specific queue, with identical arguments - without treating these as an error.
- amq_queue_39 If a bind fails, the server MUST raise a connection exception.
- amq_queue_12 The server MUST NOT allow a durable queue to bind to a transient exchange. If the client attempts this the server MUST raise a channel exception.
- amq_queue_13 Bindings for durable queues are automatically durable and the server SHOULD restore such bindings after a server restart.
- amq_queue_17 If the client attempts to an exchange that was declared as internal, the server MUST raise a connection exception with reply code 530 (not allowed).
- amq_queue_40 The server SHOULD support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.
- amq_queue_26 If the queue does not exist the server MUST raise a channel exception with reply code 404 (not found).

- amq_queue_14 If the exchange does not exist the server MUST raise a channel exception with reply code 404 (not found).

1.1.5.4 The Queue.Bind-Ok Method

This method confirms that the bind was successful.

The Bind-Ok method has the following specific fields:

This is the Bind-Ok pseudo-structure:

1.1.5.5 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 Purge method has the following specific fields:

This is the Purge pseudo-structure:

Guidelines for implementers:

- amq_queue_15 A call to purge MUST result in an empty queue.
- amq_queue_41 On transacted channels the server MUST not purge messages that have already been sent to a client but not yet acknowledged.
- amq_queue_42 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 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.
- amq_queue_16 The queue must exist. Attempting to purge a non-existing queue causes a channel exception.

1.1.5.6 The Queue.Purge-Ok Method

This method confirms the purge of a queue.

The Purge-Ok method has the following specific fields:

This is the Purge-Ok pseudo-structure:

1.1.5.7 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 cancelled.

The Delete method has the following specific fields:

This is the Delete pseudo-structure:

Guidelines for implementers:

- amq_queue_43 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.
- amq_queue_21 The queue must exist. Attempting to delete a non-existing queue causes a channel exception.
- amq_queue_29 amq_queue_30 The server MUST respect the if-unused flag when deleting a queue.

1.1.5.8 The Queue.Delete-Ok Method

This method confirms the deletion of a queue.

The Delete-Ok method has the following specific fields:

This is the Delete-Ok pseudo-structure:

1.1.6 The Basic Class

The Basic class provides methods that support an industry-standard messaging model. The ID of the Basic Class is 60.

This is the formal grammar for the class:

```
basic               = C:CONSUME S:CONSUME-OK
                    / C:CANCEL S:CANCEL-OK
                    / C:PUBLISH content
                    / S:RETURN content
                    / S:DELIVER content
                    / C:GET ( S:GET-OK content / S:GET-EMPTY )
                    / C:ACK
                    / C:REJECT
```

>These are the properties defined for $(class.name) content:

```
- content type (shortstr) -
    MIME content type
 . - content encoding (shortstr) -
    MIME content encoding
 . - headers (table) -
    Message header field table
 . - delivery mode (octet) -
    Non-persistent (1) or persistent (2)
 . - priority (octet) -
    The message priority, 0 to 9
 . - correlation id (shortstr) -
    The application correlation identifier
 . - reply to (shortstr) -
    The destination to reply to
 . - expiration (shortstr) -
    Message expiration specification
 . - message id (shortstr) -
    The application message identifier
 . - timestamp (timestamp) -
    The message timestamp
 . - type (shortstr) -
    The message type name
 . - user id (shortstr) -
    The creating user id
 . - app id (shortstr) -
    The creating application id
 . - cluster id (shortstr) -
    Intra-cluster routing identifier
 .
```

The server accepts the following methods:

- Basic.Consume (ID=10) - start a queue consumer : Sync request , carries content
- Basic.Cancel (ID=30) - end a queue consumer : Sync request
- Basic.Publish (ID=50) - publish a message : Async , carries content
- Basic.Get (ID=80) - direct access to a queue : Sync request , carries content
- Basic.Ack (ID=110) - acknowledge one or more messages : Async , carries content
- Basic.Reject (ID=120) - reject an incoming message : Async , carries content

The client accepts the following methods:

- Basic.Consume-Ok (ID=20) - confirm a new consumer : Sync response to Consume
- Basic.Cancel-Ok (ID=40) - confirm a cancelled consumer : Sync response to Cancel
- Basic.Return (ID=60) - return a failed message : Async , carries content
- Basic.Deliver (ID=70) - notify the client of a consumer message : Async , carries content
- Basic.Get-Ok (ID=90) - provide client with a message : Sync response to Get , carries content
- Basic.Get-Empty (ID=100) - indicate no messages available : Sync response to Get

1.1.6.1 The Basic.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 Consume method has the following specific fields:

This is the Consume pseudo-structure:

Guidelines for implementers:

- amq_basic_01 The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- todo The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise a connection exception with reply code 530 (not allowed).
- amq_basic_17 The server MUST ignore this setting when the client is not processing any messages - i.e. the prefetch size does not limit the transfer of single messages to a client, only the sending in advance of more messages while the client still has one or more unacknowledged messages.
- amq_basic_18 The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.
- amq_basic_02 If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise a channel exception with return code 403 (access refused).

1.1.6.2 The Basic.Consume-Ok Method

The server provides the client with a consumer tag, which is used by the client for methods called on the consumer at a later stage.

The Consume-Ok method has the following specific fields:

This is the Consume-Ok pseudo-structure:

1.1.6.3 The Basic.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 client may receive an abitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

The Cancel method has the following specific fields:

This is the Cancel pseudo-structure:

Guidelines for implementers:

- amq_basic_04
- todo If the queue no longer exists when the client sends a cancel command, or the consumer has been cancelled for other reasons, this command has no effect.

1.1.6.4 The Basic.Cancel-Ok Method

This method confirms that the cancellation was completed.

The Cancel-Ok method has the following specific fields:

This is the Cancel-Ok pseudo-structure:

1.1.6.5 The Basic.Publish Method

This method publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

The Publish method has the following specific fields:

This is the Publish pseudo-structure:

1.1.6.6 The Basic.Return Method

This method returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

The Return method has the following specific fields:

This is the Return pseudo-structure:

1.1.6.7 The Basic.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 Deliver method has the following specific fields:

This is the Deliver pseudo-structure:

Guidelines for implementers:

- amq_basic_19 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.

1.1.6.8 The Basic.Get 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 synchronous functionality is more important than performance.

The Get method has the following specific fields:

This is the Get pseudo-structure:

1.1.6.9 The Basic.Get-Ok Method

This method delivers a message to the client following a get method. A message delivered by 'get-ok' must be acknowledged unless the no-ack option was set in the get method.

The Get-Ok method has the following specific fields:

This is the Get-Ok pseudo-structure:

1.1.6.10 The Basic.Get-Empty Method

This method tells the client that the queue has no messages available for the client.

The Get-Empty method has the following specific fields:

This is the Get-Empty pseudo-structure:

1.1.6.11 The Basic.Ack Method

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

The Ack method has the following specific fields:

This is the Ack pseudo-structure:

1.1.6.12 The Basic.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 Reject method has the following specific fields:

This is the Reject pseudo-structure:

Guidelines for implementers:

- amq_basic_21 The server SHOULD be capable of accepting and process the Reject method while sending message content with a Deliver or Get-Ok method. I.e. the server should read and process incoming methods while sending output frames. To cancel a partially-send content, the server sends a content body frame of size 1 (i.e. with no data except the frame-end octet).
- amq_basic_22 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.
- amq_basic_23 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.

1.1.7 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 70.

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 content
                        / C:PUBLISH
                        / S:DELIVER
                        / S:RETURN
                        / C:ACK
                        / C:REJECT
```

>These are the properties defined for $(class.name) content:

```
- content type (shortstr) -
    MIME content type
 . - content encoding (shortstr) -
    MIME content encoding
 . - headers (table) -
    Message header field table
 . - priority (octet) -
    The message priority, 0 to 9
 . - reply to (shortstr) -
    The destination to reply to
 . - message id (shortstr) -
    The application message identifier
 . - filename (shortstr) -
    The message filename
 . - timestamp (timestamp) -
    The message timestamp
 . - cluster id (shortstr) -
    Intra-cluster routing identifier
 .
```

The server accepts the following methods:

- File.Consume (ID=10) - start a queue consumer : Sync request , carries content
- File.Cancel (ID=30) - end a queue consumer : Sync request
- File.Open (ID=50) - request to start staging : Sync request , carries content
- File.Open-Ok (ID=60) - confirm staging ready : Sync request
- File.Stage (ID=70) - stage message content : Async
- File.Publish (ID=80) - publish a message : Async , carries content
- File.Ack (ID=110) - acknowledge one or more messages : Async , carries content
- File.Reject (ID=120) - reject an incoming message : Async , carries content

The client accepts the following methods:

- File.Consume-Ok (ID=20) - confirm a new consumer : Sync response to Consume
- File.Cancel-Ok (ID=40) - confirm a cancelled consumer : Sync response to Cancel
- File.Open (ID=50) - request to start staging : Sync request , carries content
- File.Open-Ok (ID=60) - confirm staging ready : Sync request

- File.Stage (ID=70) - stage message content : Async
- File.Return (ID=90) - return a failed message : Async , carries content
- File.Deliver (ID=100) - notify the client of a consumer message : Async , carries content

1.1.7.1 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 Consume method has the following specific fields:

This is the Consume pseudo-structure:

Guidelines for implementers:

- The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- todo The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise a connection exception with reply code 530 (not allowed).
- The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.
- amq_file_00 If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise a channel exception with return code 405 (resource locked).

1.1.7.2 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 Consume-Ok method has the following specific fields:

This is the Consume-Ok pseudo-structure:

1.1.7.3 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 Cancel method has the following specific fields:

This is the Cancel pseudo-structure:

1.1.7.4 The File.Cancel-Ok Method

This method confirms that the cancellation was completed.

The Cancel-Ok method has the following specific fields:

This is the Cancel-Ok pseudo-structure:

1.1.7.5 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 Open method has the following specific fields:

This is the Open pseudo-structure:

1.1.7.6 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 Open-Ok method has the following specific fields:

This is the Open-Ok pseudo-structure:

1.1.7.7 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.

The Stage method has the following specific fields:

This is the Stage pseudo-structure:

1.1.7.8 The File.Publish Method

This method publishes a staged file message to a specific exchange. The file message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

The Publish method has the following specific fields:

This is the Publish pseudo-structure:

1.1.7.9 The File.Return Method

This method returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

The Return method has the following specific fields:

This is the Return pseudo-structure:

1.1.7.10 The File.Deliver Method

This method delivers a staged file 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 Deliver method has the following specific fields:

This is the Deliver pseudo-structure:

Guidelines for implementers:

- 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.

1.1.7.11 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 Ack method has the following specific fields:

This is the Ack pseudo-structure:

1.1.7.12 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 Reject method has the following specific fields:

This is the Reject pseudo-structure:

Guidelines for implementers:

- 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 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.

1.1.8 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 80.

This is the formal grammar for the class:

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

>These are the properties defined for $(class.name) content:

```
- content type (shortstr) -
   MIME content type
 . - content encoding (shortstr) -
   MIME content encoding
 . - headers (table) -
   Message header field table
 . - priority (octet) -
   The message priority, 0 to 9
 . - timestamp (timestamp) -
   The message timestamp
 .
```

The server accepts the following methods:

- Stream.Consume (ID=10) - start a queue consumer : Sync request , carries content
- Stream.Cancel (ID=30) - end a queue consumer : Sync request
- Stream.Publish (ID=50) - publish a message : Async , carries content

The client accepts the following methods:

- Stream.Consume-Ok (ID=20) - confirm a new consumer : Sync response to Consume
- Stream.Cancel-Ok (ID=40) - confirm a cancelled consumer : Sync response to Cancel
- Stream.Return (ID=60) - return a failed message : Async , carries content
- Stream.Deliver (ID=70) - notify the client of a consumer message : Async , carries content

1.1.8.1 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 Consume method has the following specific fields:

This is the Consume pseudo-structure:

Guidelines for implementers:

- The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.
- Streaming applications SHOULD use different channels to select different streaming resolutions. AMQP makes no provision for filtering and/or transforming streams except on the basis of priority-based selective delivery of individual messages.
- The client MUST provide a valid access ticket giving "read" access rights to the realm for the queue.
- "todo" The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise a connection exception with reply code 530 (not allowed).
- 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.
- amq_file_00 If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise a channel exception with return code 405 (resource locked).

1.1.8.2 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 Consume-Ok method has the following specific fields:

This is the Consume-Ok pseudo-structure:

1.1.8.3 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 canceling a consumer. It may process or discard these as appropriate.

The Cancel method has the following specific fields:

This is the Cancel pseudo-structure:

1.1.8.4 The Stream.Cancel-Ok Method

This method confirms that the cancellation was completed.

The Cancel-Ok method has the following specific fields:

This is the Cancel-Ok pseudo-structure:

1.1.8.5 The Stream.Publish Method

This method publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers as appropriate.

The Publish method has the following specific fields:

This is the Publish pseudo-structure:

1.1.8.6 The Stream.Return Method

This method returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

The Return method has the following specific fields:

This is the Return pseudo-structure:

1.1.8.7 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 Deliver method has the following specific fields:

This is the Deliver pseudo-structure:

1.1.9 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 90.

This is the formal grammar for the class:

```
tx                  = C:SELECT S:SELECT-OK
                    / C:COMMIT S:COMMIT-OK
                    / C:ROLLBACK S:ROLLBACK-OK
```

The server accepts the following methods:

- Tx.Select (ID=10) - select standard transaction mode : Sync request
- Tx.Commit (ID=30) - commit the current transaction : Sync request
- Tx.Rollback (ID=50) - abandon the current transaction : Sync request

The client accepts the following methods:

- Tx.Select-Ok (ID=20) - confirm transaction mode : Sync response to Select
- Tx.Commit-Ok (ID=40) - confirm a successful commit : Sync response to Commit
- Tx.Rollback-Ok (ID=60) - confirm a successful rollback : Sync response to Rollback

1.1.9.1 The Tx.Select Method

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

The Select method has the following specific fields:

This is the Select pseudo-structure:

1.1.9.2 The Tx.Select-Ok Method

This method confirms to the client that the channel was successfully set to use standard transactions.

The Select-Ok method has the following specific fields:

This is the Select-Ok pseudo-structure:

1.1.9.3 The Tx.Commit Method

This method commits all messages published and acknowledged in the current transaction. A new transaction starts immediately after a commit.

The Commit method has the following specific fields:

This is the Commit pseudo-structure:

1.1.9.4 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.

The Commit-Ok method has the following specific fields:

This is the Commit-Ok pseudo-structure:

1.1.9.5 The Tx.Rollback Method

This method abandons all messages published and acknowledged in the current transaction. A new transaction starts immediately after a rollback.

The Rollback method has the following specific fields:

This is the Rollback pseudo-structure:

1.1.9.6 The Tx.Rollback-Ok Method

This method confirms to the client that the rollback succeeded. Note that if an rollback fails, the server raises a channel exception.

The Rollback-Ok method has the following specific fields:

This is the Rollback-Ok pseudo-structure:

1.1.10 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 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) to a distributed transaction coordinator. The ID of the Dtx Class is 100.

This is the formal grammar for the class:

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

The server accepts the following methods:

- Dtx.Select (ID=10) - select standard transaction mode : Sync request
- Dtx.Start (ID=30) - start a new distributed transaction : Sync request

The client accepts the following methods:

- Dtx.Select-Ok (ID=20) - confirm transaction mode : Sync response to Select
- Dtx.Start-Ok (ID=40) - confirm the start of a new distributed transaction : Sync response to Start

1.1.10.1 The Dtx.Select Method

This method sets the channel to use distributed transactions. The client must use this method at least once on a channel before using the Start method.

The Select method has the following specific fields:

This is the Select pseudo-structure:

1.1.10.2 The Dtx.Select-Ok Method

This method confirms to the client that the channel was successfully set to use distributed transactions.

The Select-Ok method has the following specific fields:

This is the Select-Ok pseudo-structure:

1.1.10.3 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 Start method has the following specific fields:

This is the Start pseudo-structure:

1.1.10.4 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.

The Start-Ok method has the following specific fields:

This is the Start-Ok pseudo-structure:

1.1.11 The Tunnel Class

The tunnel methods are used to send blocks of binary data - which can be serialised AMQP methods or other protocol frames - between AMQP peers. The ID of the Tunnel Class is 110.

This is the formal grammar for the class:

```
   tunnel              = C:REQUEST
                       / S:REQUEST
```

>These are the properties defined for $(class.name) content:

```
- headers (table) -
    Message header field table
 . - proxy name (shortstr) -
    The identity of the tunnelling proxy
 . - data name (shortstr) -
    The name or type of the message being tunnelled
 . - durable (octet) -
    The message durability indicator
 . - broadcast (octet) -
    The message broadcast mode
 .
```

The server accepts the following methods:

- Tunnel.Request (ID=10) - sends a tunnelled method : Async

The client accepts the following methods:

1.1.11.1 The Tunnel.Request Method

This method tunnels a block of binary data, which can be an encoded AMQP method or other data. The binary data is sent as the content for the Tunnel.Request method.

The Request method has the following specific fields:

This is the Request pseudo-structure:

1.1.12 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 120.

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
```

The server accepts the following methods:

- Test.Integer (ID=10) - test integer handling : Sync request , carries content
- Test.Integer-Ok (ID=20) - report integer test result : Sync response to Integer
- Test.String (ID=30) - test string handling : Sync request , carries content
- Test.String-Ok (ID=40) - report string test result : Sync response to String
- Test.Table (ID=50) - test field table handling : Sync request , carries content
- Test.Table-Ok (ID=60) - report table test result : Sync response to Table , carries content
- Test.Content (ID=70) - test content handling : Sync request
- Test.Content-Ok (ID=80) - report content test result : Sync response to Content

The client accepts the following methods:

- Test.Integer (ID=10) - test integer handling : Sync request , carries content
- Test.Integer-Ok (ID=20) - report integer test result : Sync response to Integer
- Test.String (ID=30) - test string handling : Sync request , carries content
- Test.String-Ok (ID=40) - report string test result : Sync response to String
- Test.Table (ID=50) - test field table handling : Sync request , carries content
- Test.Table-Ok (ID=60) - report table test result : Sync response to Table , carries content
- Test.Content (ID=70) - test content handling : Sync request
- Test.Content-Ok (ID=80) - report content test result : Sync response to Content

1.1.12.1 The Test.Integer Method

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

The Integer method has the following specific fields:

This is the Integer pseudo-structure:

1.1.12.2 The Test.Integer-Ok Method

This method reports the result of an Integer method.

The Integer-Ok method has the following specific fields:

This is the Integer-Ok pseudo-structure:

1.1.12.3 The Test.String Method

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

The String method has the following specific fields:

This is the String pseudo-structure:

1.1.12.4 The Test.String-Ok Method

This method reports the result of a String method.

The String-Ok method has the following specific fields:

This is the String-Ok pseudo-structure:

1.1.12.5 The Test.Table Method

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

The Table method has the following specific fields:

This is the Table pseudo-structure:

1.1.12.6 The Test.Table-Ok Method

This method reports the result of a Table method.

The Table-Ok method has the following specific fields:

This is the Table-Ok pseudo-structure:

1.1.12.7 The Test.Content Method

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

The Content method has the following specific fields:

This is the Content pseudo-structure:

1.1.12.8 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 Content-Ok method has the following specific fields:

This is the Content-Ok pseudo-structure:

1.1.13 The Cluster Class

The cluster methods are used by peers in a cluster. The ID of the Cluster Class is 61440.

This is the formal grammar for the class:

```
cluster             = C:HELLO
                    / C:STATUS
                    / C:BIND
```

The server accepts the following methods:

- Cluster.Hello (ID=10) - greet cluster peer : Async , carries content
- Cluster.Status (ID=20) - provide peer status data : Async , carries content
- Cluster.Bind (ID=30) - bind local exchange to remote exchange : Async , carries content

The client accepts the following methods:

- Cluster.Hello (ID=10) - greet cluster peer : Async , carries content
- Cluster.Status (ID=20) - provide peer status data : Async , carries content
- Cluster.Bind (ID=30) - bind local exchange to remote exchange : Async , carries content

1.1.13.1 The Cluster.Hello Method

This method tells the cluster peer our name and cluster protocol version.

The Hello method has the following specific fields:

This is the Hello pseudo-structure:

1.1.13.2 The Cluster.Status Method

This method provides a cluster peer with status information. We use this method for cluster heartbeating and synchronisation.

The Status method has the following specific fields:

This is the Status pseudo-structure:

1.1.13.3 The Cluster.Bind Method

This method binds an exchange on one server to an exchange on another server.

The Bind method has the following specific fields:

This is the Bind pseudo-structure: